

4th USENIX Conference on File and Storage Technologies

*San Francisco, CA, USA
December 13–16, 2005*

Sponsored by
The USENIX Association
in cooperation with
**ACM SIGOPS, IEEE Mass Storage Systems
Technical Committee (MSSTC), and
IEEE TCOS**

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: <http://www.usenix.org>

The price is \$35 for members and \$45 for nonmembers.
Outside the U.S.A. and Canada, please add
\$15 per copy for postage (via air printed matter).

Past FAST Proceedings

FAST '04	March 31–April 2, 2004	San Francisco, California, USA	\$30/\$40
FAST '03	March 31–April 2, 2003	San Francisco, California, USA	\$30/\$40
FAST '02	January 28–30, 2002	Monterey, California, USA	\$25/\$32

© 2005 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-39-0

USENIX Association

**Proceedings of the 4th
USENIX Conference on
File and Storage Technologies**

**December 13–16, 2005
San Francisco, CA, USA**

Conference Organizers

Program Chair

Garth Gibson, *Carnegie Mellon University and Panasas*

Program Committee

Andrea Arpaci-Dusseau, *University of Wisconsin, Madison*

David L. Black, *EMC Corporation*

Peter Corbett, *Network Appliance*

Daniel Ellard, *Sun Microsystems Laboratories*

Jason Flinn, *University of Michigan, Ann Arbor*

Richard Golding, *IBM Almaden*

Peter Honeyman, *CITI, University of Michigan, Ann Arbor*

Jon Howell, *Microsoft Research*

Kimberly Keeton, *Hewlett-Packard Labs*

John Kubiawicz, *University of California, Berkeley*

Kai Li, *Princeton University and Data Domain*

Darrell Long, *University of California, Santa Cruz*

Robert Morris, *Massachusetts Institute of Technology*

Dick Muntz, *University of California, Los Angeles*

Ian Pratt, *University of Cambridge, UK*

Rod Van Meter, *Keio University, Japan*

Steering Committee

Jeff Chase, *Duke University*

Jack Cole, *Army Research Lab*

Greg Ganger, *Carnegie Mellon University*

Garth Gibson, *Carnegie Mellon University and Panasas*

Peter Honeyman, *CITI, University of Michigan, Ann Arbor*

John Howard, *Sun Microsystems*

Merritt Jones, *MITRE Corporation*

Darrell Long, *University of California, Santa Cruz*

Jai Menon, *IBM Research*

Margo Seltzer, *Harvard University*

John Wilkes, *Hewlett-Packard Labs*

Ellie Young, *USENIX Association*

The USENIX Association Staff

External Reviewers

Atul Adya	Angela Demke	Windsor Hsu	Pankaj Mehra	Alma Riska	Nisha Talagala
Marcos Aguilera	Brown	Lan Huang	Arif Merchant	Ohad Rodeh	Pat Thaler
Guillermo Alvarez	Timothy Denehy	James Hughes	Mike Mesnier	Robert Ross	Marvin Theimer
Ahmed Amer	John Douceur	Michael Isard	Joerg Meyer	Russ Ross	Chandu Thekkath
Khalil Amiri	Boris Dragovic	Arkady Kanevsky	Ethan Miller	Thomas Ruwart	Alexander Thomasian
David Andersen	Patrick Eaton	Brent Kang	Tim Moreton	Brandon Salmon	Jim Thornton
Eric Anderson	Ted Faber	Christos Karamanolis	Kiran-Kumar	Jose Renato Santos	Niraj Tolia
Ismail Ari	Jered Floyd	Michael Kazar	Muniswamy-Reddy	Douglas Santry	Yoshio Turner
Remzi Arpaci-Dusseau	Richard Freitas	Deepak	Athicha	Prasenjit Sarkar	Mustafa Uysal
Mary Baker	Archana Ganapathi	Kenchammana	Muthitacharoen	Julian Satran	Amin Vahdat
Ralph Becker-Szendy	Greg Ganger	Olga Kornievskaya	Trond Myklebust	Mahadev	Catharine van Ingen
Frank Belloso	Robert Garner	Pekka Kostamaa	David Nagle	Satyanarayanan	Kaladhar Voruganti
Sanjit Biswas	Morrie Gasaser	Orran Krieger	Dushyanth Narayanan	Jiri Schindler	Rosie Wacha
Bill Bolosky	Shahram	Max Krohn	Ed Nightingale	Frank Schmuck	An-I Wang
Elizabeth Borowsky	Ghandeharizadeh	Geoff Kuenning	Brian Noble	Thomas Schwarz	Helen Wang
Aaron Brown	Jonathon Giffin	Zachary Kurmas	Emil Ong	Gauri Shah	Andrew Warfield
Jehoshua Bruck	Thomer Gil	Jonathan Ledlie	D. Stott Parker	Mehul Shah	Hakim Weatherspoon
Nicholas Burke	Atul Goel	Ed Lee	Shankar Pasupathy	Cyrus Shahabi	Sage Weil
Randal Burns	Garth Goodson	Chris Lesniewski	Hugo Patterson	Marc Shapiro	Brent Welch
Pei Cao	Tomislav Gracanac	Laas	Daniel Peek	Emil Sit	Ric Wheeler
Mark Carlson	Stephen Green	Charles Lever	David Petrou	Muthian Sivathanu	John Wilkes
Jeff Chase	Kevin Greenan	Jinyang Li	James Plank	Christopher Small	Theodore Wong
Alvin Chen	James Hafner	Caixue Lin	Kristal Pollack	Keith Smith	Qin Xin
Peter Chen	Benny Halevy	Jane Loizeaux	Florentina Popovici	Craig Soules	Frank Tsozen Yeh
Ying Chen	Steven Hand	Jacob Lorch	BJ Premore	Edgar St.Pierre	Alexander Yip
Byung-Gon Chun	John Hartman	John C.S. Lui	Seth Proctor	David Steere	Erez Zadok
Mark Corner	Jon Haswell	Christopher Lumb	KK Rao	Lex Stein	Carlo Zaniolo
Landon Cox	Val Henson	Qin Lv	Narasimha Reddy	Daniel Stodolsky	Chi Zhang
Russ Cox	Eric Hibbard	John MacCormick	Ben Reed	Jeremy Stribling	Jiaying Zhang
Roger Cummings	Mark Holland	Umesh Maheshwari	Mike Reiter	John Strunk	Yuanyuan Zhou
Frank Dabek	Bo Hong	Dahlia Malkhi	Sean Rhea	Darrell Suggs	Ningning Zhu
Mike Dahlin	Andy Hospodor	Carlos Maltzahn	Wayne Rickard	Sai Susarla	
	John Howard	Timothy Mann	Erik Riedel	Ram Swaminathan	

4th USENIX Conference on File and Storage Technologies

December 13–16, 2005

San Francisco, CA, USA

Index of Authors	v
Message from the Program Chair	vii

Wednesday, December 14, 2005

File Systems Semantics

A Logic of File Systems	1
<i>Muthian Sivathanu, Google Inc.; Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha, University of Wisconsin, Madison</i>	
Providing Tunable Consistency for a Parallel File Store	17
<i>Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam, Pennsylvania State University</i>	

Sensor Storage

MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices	31
<i>Demitrios Zeinalipour-Yazti, University of Cyprus; Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar, University of California, Riverside</i>	
Adaptive Data Placement for Wide-Area Sensing Services	45
<i>Suman Nath, Microsoft Research; Phillip B. Gibbons, Intel Research Pittsburgh; Srinivasan Seshan, Carnegie Mellon University</i>	

Fault Handling

Ursa Minor: Versatile Cluster-based Storage	59
<i>Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John D. Strunk, Eno Thereska, Matthew Wachs, and Jay J. Wylie, Carnegie Mellon University</i>	
Zodiac: Efficient Impact Analysis for Storage Area Networks	73
<i>Aameek Singh, Georgia Institute of Technology; Madhukar Korupolu and Kaladhar Voruganti, IBM Almaden Research Center</i>	
Journal-guided Resynchronization for Software RAID	87
<i>Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison</i>	

Caching

DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality	101
<i>Song Jiang, Los Alamos National Laboratory; Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang, Ohio State University</i>	
Second-Tier Cache Management Using Write Hints	115
<i>Xuhui Li, Ashraf Aboulmaga, and Kenneth Salem, University of Waterloo; Aamer Sachedina, IBM Toronto Lab; Shaobo Gao, University of Waterloo</i>	
WOW: Wise Ordering for Writes—Combining Spatial and Temporal Locality in Non-Volatile Caches	129
<i>Binny S. Gill and Dharmendra S. Modha, IBM Almaden Research Center</i>	

Thursday, December 15, 2005

Security

Secure Deletion for a Versioning File System 143
Zachary N.J. Peterson, Randal Burns, Joe Herring, Adam Stubblefield, and Aviel D. Rubin, The Johns Hopkins University

TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study 155
Jinpeng Wei and Calton Pu, Georgia Institute of Technology

A Security Model for Full-Text File System Search in Multi-User Environments 169
Stefan Büttcher and Charles L. A. Clarke, University of Waterloo

Multi-Fault Tolerance

Matrix Methods for Lost Data Reconstruction in Erasure Codes 183
James Lee Hafner, Veera Deenadhayalan, and KK Rao, IBM Almaden Research Center; John A. Tomlin, Yahoo! Research

STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures 197
Cheng Huang, Microsoft Research; Lihao Xu, Wayne State University

WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems 211
James Lee Hafner, IBM Almaden Research Center

On the Media

On Multidimensional Data and Modern Disks 225
Steven W. Schlosser, Intel Research Pittsburgh; Jiri Schindler, EMC Corporation; Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger, Carnegie Mellon University

Database-Aware Semantically-Smart Storage 239
Muthian Sivathanu, Google Inc.; Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, University of Wisconsin, Madison

Managing Prefetch Memory for Data-Intensive Online Servers 253
Chuanpeng Li and Kai Shen, University of Rochester

Friday, December 16, 2005

On the Wire

A Scalable and High Performance Software iSCSI Implementation 267
Abhijeet Joglekar, Michael E. Kounavis, and Frank L. Berry, Intel Research and Development

TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization 281
Navendu Jain and Mike Dahlin, University of Texas at Austin; Renu Tewari, IBM Almaden Research Center

VXA: A Virtual Architecture for Durable Compressed Archives 295
Bryan Ford, CSAIL, Massachusetts Institute of Technology

Tools

I/O System Performance Debugging Using Model-driven Anomaly Characterization 309
Kai Shen, Ming Zhong, and Chuanpeng Li, University of Rochester

TBBT: Scalable and Accurate Trace Replay for File Server Evaluation 323
Ningning Zhu, Jiawu Chen, and Tzi-cker Chiueh, Stony Brook University

Accurate and Efficient Replaying of File System Traces 337
Nikolai Joukov, Timothy Wong, and Erez Zadok, Stony Brook University

Index of Authors

Abd-El-Malek, Michael	59	Mesnier, Michael	59
Aboulnaga, Ashraf	115	Modha, Dharmendra S.	129
Ailamaki, Anastassia	225	Najjar, Walid A.	31
Arpaci-Dusseau, Andrea C.	1, 87, 239	Nath, Partho	17
Arpaci-Dusseau, Remzi H.	1, 87, 239	Nath, Suman	45
Bairavasundaram, Lakshmi N.	239	Papadomanolakis, Stratos	225
Berry, Frank L.	267	Peterson, Zachary N.J.	143
Burns, Randal	143	Prasad, Manish	59
Büttcher, Stefan	169	Pu, Calton	155
Chen, Feng	101	Rao, KK	183
Chen, Jiawu	323	Rubin, Aviel D.	143
Chiueh, Tzi-cker	323	Sachedina, Aamer	115
Clarke, Charles L. A.	169	Salem, Kenneth	115
Courtright II, William V.	59	Salmon, Brandon	59
Cranor, Chuck	59	Sambasivan, Raja R.	59
Dahlin, Mike	281	Schindler, Jiri	225
Deenadhayalan, Veera	183	Schlosser, Steven W.	225
Denehy, Timothy E.	87	Seshan, Srinivasan	45
Ding, Xiaoning	101	Shao, Minglong	225
Faloutsos, Christos	225	Shen, Kai	253, 309
Ford, Bryan	295	Singh, Aameek	73
Ganger, Gregory R.	59, 225	Sinnamohideen, Shafeeq	59
Gao, Shaobo	115	Sivasubramaniam, Anand	17
Gibbons, Phillip B.	45	Sivathanu, Muthian	1, 239
Gill, Binny S.	129	Strunk, John D.	59
Gunopulos, Dimitrios	31	Stubblefield, Adam	143
Hafner, James Lee	183, 211	Tan, Enhua	101
Hendricks, James	59	Tewari, Renu	281
Herring, Joe	143	Thereska, Eno	59
Huang, Cheng	197	Tomlin, John A.	183
Jain, Navendu	281	Vilayannur, Murali	17
Jha, Somesh	1	Voruganti, Kaladhar	73
Jiang, Song	101	Wachs, Matthew	59
Joglekar, Abhijeet	267	Wei, Jinpeng	155
Joukov, Nikolai	337	Wong, Timothy	337
Kalogeraki, Vana	31	Wylie, Jay J.	59
Klosterman, Andrew J.	59	Xu, Lihao	197
Korupolu, Madhukar	73	Zadok, Erez	337
Kounavis, Michael E.	267	Zeinalipour-Yazti, Demtrios	31
Li, Chuanpeng	253, 309	Zhang, Xiaodong	101
Li, Xuhui	115	Zhong, Ming	309
Lin, Song	31	Zhu, Ningning	323

Message from the Program Chair

I am pleased to present the program for the 4th USENIX Conference on File and Storage Technologies (FAST '05). This conference strives to bring together storage system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. Each of the papers of this year's single-track program were presented in San Francisco, December 14–16, 2005. This program includes topics ranging from logic models for file systems to implementation techniques for high-speed software iSCSI and from storage systems for flash-based sensor devices to correcting more than two simultaneously failed disks.

New to FAST, the 4th FAST conference included a day of tutorials. Held on December 13, 2005, four tutorials were offered: erasure codes for storage applications, object-based cluster storage systems, the inside story of NFS version 4, and a SNIA-sponsored tutorial on SAS, SATA, Fibre Channel, and iSCSI storage protocols.

In cooperation and co-located with FAST '05 was the 3rd International IEEE Security in Storage Workshop, also held on December 13, 2005. I encourage everyone in the file and storage research community to expand their knowledge and interest in storage security.

The paper selection for FAST '05 was quite competitive. The number of submitted papers, 125, exceeds any of the prior FAST conferences. Of these 125 submissions, 25 papers (20%) were selected. Submitted papers were blind, i.e., reviewers did not know the identities of a submission's authors, and the program committee was able to make almost all selection decisions without learning author identities. The program committee also made a careful effort not to ask for reviews from conflicted people, where "conflict" was broadly defined: A reviewer should not have been affiliated with any author's primary institution or have participated in collaborative research with any author in the past five years, and should not review the work of a Ph.D. advisor or Ph.D. student. Most submissions received five or more reviews, typically three of which came from the 16-member program committee.

The program committee met on August 26, 2005. The committee discussed all submissions ranking in the top 50 by overall average merit and average with the lowest score excluded, plus all submissions with a high variance on the overall merit scores, those with a low number of reviews, and all low-scoring submissions advocated by any program committee member. Program committee members stepped out of the meeting during the discussion of any submission whose authors were in conflict with the PC member. Papers co-authored by a PC member were held to a higher standard. There were only three papers accepted with PC co-authors.

I would like first to thank the authors of all submitted papers. The success of a conference is first and foremost a result of the high quality of its submissions. Next, I would like to thank the external reviewers for their diligence and fairness on all reviews. Of course, the program committee members carried the majority of the burden for fairly reviewing submissions and selecting accepted papers: Andrea Arpaci-Dusseau, David L. Black, Peter Corbett, Daniel Ellard, Jason Flinn, Richard Golding, Peter Honeyman, Jon Howell, Kimberly Keeton, John Kubiawicz, Kai Li, Darrell Long, Robert Morris, Dick Muntz, Ian Pratt, and Rod Van Meter. Their diligence in this task and helpful advice during the process were essential and greatly appreciated by me. For advice and guidance in the selection of the program committee and the procedures it used, I am grateful for the help of the FAST steering committee: Jeff Chase, Jack Cole, Greg Ganger, Peter Honeyman, John Howard, Merritt Jones, Darrell Long, Jai Menon, Margo Seltzer, John Wilkes, and Ellie Young. I would also like to thank Dirk Grunwald for developing the CRP reviewing package FAST '05 employed.

Finally, I would like to thank the administrative folks who provided the really essential help needed to pull together this program. Stan Bielski was indispensable as the Webmaster for the reviewing site. Angela Miller and Karen Lindenfelser managed arrangements and logistics flawlessly. At USENIX, Dan Klein guided the construction of FAST's new tutorial offering, Anne Dickison managed the marketing for FAST '05, Jane-Ellen Long managed the publications needs for FAST '05, and, of course, Ellie Young managed us all with grace and effectiveness.

Enjoy the program!

**Garth Gibson, Carnegie Mellon University and Panasas Inc.
FAST '05 Program Chair**

A Logic of File Systems

Muthian Sivathanu*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Somesh Jha
Google Inc. Computer Sciences Department, University of Wisconsin, Madison

muthian@google.com, {dusseau, remzi, jha}@cs.wisc.edu

Abstract

Years of innovation in file systems have been highly successful in improving their performance and functionality, but at the cost of complicating their interaction with the disk. A variety of techniques exist to ensure consistency and integrity of file system data, but the precise set of correctness guarantees provided by each technique is often unclear, making them hard to compare and reason about. The absence of a formal framework has hampered detailed verification of file system correctness.

We present a logical framework for modeling the interaction of a file system with the storage system, and show how to apply the logic to represent and prove correctness properties. We demonstrate that the logic provides three main benefits. First, it enables reasoning about existing file system mechanisms, allowing developers to employ aggressive performance optimizations without fear of compromising correctness. Second, the logic simplifies the introduction and adoption of new file system functionality by facilitating rigorous proof of their correctness. Finally, the logic helps reason about smart storage systems that track semantic information about the file system.

A key aspect of the logic is that it enables *incremental modeling*, significantly reducing the barrier to entry in terms of its actual use by file system designers. In general, we believe that our framework transforms the hitherto esoteric and error-prone “art” of file system design into a readily understandable and formally verifiable process.

1 Introduction

Reliable data storage is the cornerstone of modern computer systems. File systems are responsible for managing persistent data, and it is therefore essential to ensure that they function correctly.

Unfortunately, modern file systems have evolved into extremely complex pieces of software, incorporating sophisticated performance optimizations and features. Because disk I/O is the key bottleneck in file system performance, most optimizations aim at minimizing disk access, often at the cost of complicating the interaction of the file system with the storage system; while early file systems adopted simple update policies that were easy to reason about [11], modern file systems have significantly more complex interaction with the disk, mainly stemming from asynchrony in updates to metadata [2, 6, 8, 12, 18, 22, 23].

Reasoning about the interaction of a file system with disk is paramount to ensuring that the file system never corrupts or loses data. However, with complex update policies, the precise set of guarantees that the file system provides is obscured, and reasoning about its behavior often translates into a manual intuitive exploration of various scenarios by the developers; such *ad hoc* exploration is arduous [23], and possibly error-prone. For example, recent work [24] has found major correctness errors in widely used file systems such as ext3, ReiserFS and JFS.

In this paper, we present a formal logic for modeling the interaction of a file system with the disk. With formal modeling, we show that reasoning about file system correctness is simple and foolproof. The need for such a formal model is illustrated by the existence of similar frameworks in many other areas where correctness is paramount; existing models for authentication protocols [4], database reliability [7], and database recovery [9] are a few examples. While general theories for modeling concurrent systems exist [1, 10], such frameworks are too general to model file systems effectively; a domain-specific logic greatly simplifies modeling [4].

A logic of file systems serves three important purposes. First, it enables us to prove properties about existing file system designs, resulting in better understanding of the set of guarantees and enabling aggressive performance optimizations that preserve those guarantees. Second, it significantly lowers the barrier to providing new mechanisms or functionality in the file system by enabling rigorous reasoning about their correctness; in the absence of such a framework, designers tend to stick with “time-tested” alternatives. Finally, the logic helps design functionality in new class of storage systems [20] by facilitating precise characterization and proof of their properties.

A key goal of the logic framework is *simplicity*; in order to be useful to general file system designers, the barrier to entry in terms of applying the logic should be low. Our logic achieves this by enabling *incremental modeling*. One need not have a complete model of a file system before starting to use the logic; instead, one can simply model a particular piece of functionality or mechanism in isolation and prove properties about it.

Through case studies, we demonstrate the utility and efficacy of our logic in reasoning about file system cor-

*Work done while at the University of Wisconsin-Madison

rectness properties. First, we represent and prove the soundness of important guarantees provided by existing techniques for file system consistency, such as soft updates and journaling. We then use the logic to prove that the Linux ext3 file system is needlessly conservative in its transaction commits, resulting in sub-optimal performance; this case study demonstrates the utility of the logic in enabling aggressive performance optimizations.

To illustrate the utility of the logic in developing new file system functionality, we propose a new file system mechanism called *generation pointers* to enable *consistent undelete* of files. We prove the correctness of our design by incremental modeling of this mechanism in our logic, demonstrating the simplicity of the process. We then implement the mechanism in the Linux ext3 file system, and verify its correctness. As the logic indicates, we empirically show that inconsistency does indeed occur in undeletes in the absence of our mechanism.

The rest of the paper is organized as follows. We first present an extended motivation (§2), and a background on file systems (§3). We present the basic entities in our logic (§4) and the formalism (§5), and represent some common file system properties using the logic (§6). We then use the logic to prove consistency properties of existing systems (§7), prove the correctness of an unexploited performance optimization in ext3 (§8), and reason about a new technique for consistent undeletes (§9). We then apply our logic to semantic disks (§10). Finally, we present related work (§11) and conclude (§12).

2 Extended Motivation

A systematic framework for reasoning about the interaction of a file system with the disk has multifarious benefits. We describe three key applications of the framework.

2.1 Reasoning about existing file systems

An important usage scenario for the logic is to model existing file systems. There are three key benefits to such modeling. First, it enables a clear understanding of the precise guarantees that a given mechanism provides, and the assumptions under which those guarantees hold. Such an understanding enables correct implementation of functionality at *other* system layers such as the disk system by ensuring that they do not adversely interact with the file system assumptions. For example, write-back caching in disks often results in reordering of writes to the media; this can negate the assumptions journaling is based on.

Second, the logic enables aggressive performance optimizations. When reasoning about complex interactions becomes hard, file system developers tend to be conservative (*e.g.*, perform unnecessarily more waits). Our logic helps remove this barrier, enabling developers to be aggressive in their performance optimizations while still being confident of their correctness. In Section 8, we analyze a real example of such an opportunity for optimization

in the Linux ext3 file system, and show that the logic framework can help prove its correctness.

The final benefit of the logic framework is its potential use in implementation-level model checkers [24]; having a clear model of expected behavior against which to validate an existing file system would perhaps enable more comprehensive and efficient model checking, instead of the current technique of relying on the *fsck* mechanism which is quite expensive; the cost of an *fsck* on every explored state limits the scalability of such model checking.

2.2 Building new file system functionality

Recovery and consistency are traditionally viewed as “tricky” issues to reason about and get right. A classic illustration of this view arises in database recovery; the widely used ARIES [13] algorithm pointed to correctness issues with many earlier proposals. Ironically, the success of ARIES stalled innovation in database recovery, due to the difficulty in proving the correctness of new techniques.

Given that most innovation within the file system deals with its interaction with the disk and can have correctness implications, this inertia against changing “time-tested” alternatives stifles the incorporation of new functionality in file systems. A systematic framework to reason about a new piece of functionality can greatly reduce this barrier to entry. In Section 9, we propose new file system functionality and use our logic to prove its correctness. To further illustrate the efficacy of the logic in reasoning about new functionality, we examine in Section 7.2.1 a common file system feature, *i.e.*, journaling, and show that starting from a simple logical model of journaling, we can systematically arrive at the various corner cases that need to be handled, some of which involve complex interactions as described by the developers of Linux Ext3 [23].

2.3 Designing semantically-smart disks

The logic framework also significantly simplifies reasoning about a new class of storage systems called *semantically-smart disk systems* that provide enhanced functionality by inferring file system operations [20]. Inferring information accurately underneath modern file systems is known to be quite complex [21], especially because it is dependent on dynamic file system properties. In Section 10, we show that the logic can simplify reasoning about a semantic disk; this can in turn enable aggressive functionality in them.

3 Background

A file system organizes disk blocks into logical files and directories. In order to map blocks to logical entities such as files, the file system tracks various forms of *metadata*. In this section, we first describe the forms of metadata that file systems track, and then discuss the issue of file system consistency. Finally, we describe the asynchrony of file

systems, a major source of complexity in its interaction with disk.

3.1 File system metadata

File system metadata can be classified into three types:

Directories: Directories map a logical file name to per-file metadata. Since the file mapped for a name can be a directory itself, directories enable a hierarchy of files. When a user opens a file specifying its *path name*, the file system locates the per-file metadata for the file, reading each directory in the path if required.

File metadata: File metadata contains information about a specific file. Examples of such information are the set of disk blocks that comprise the file, file size, and so on. In certain file systems such as FAT, file metadata is embedded in the directory entries, while in most other file systems, file metadata is stored separately (*e.g.*, inodes) and is pointed to by the directory entries. The pointers from file metadata to the disk blocks can sometimes be indirected through *indirect pointer* blocks in the case of large files.

Allocation structures: File systems manage various resources on disk such as the set of free blocks that can be allocated to new files. To track such resources, file systems maintain structures (*e.g.*, bitmaps, free lists) that point to free resource instances.

In addition, file systems track other metadata (*e.g.*, super block), but we mainly focus on the above three types.

3.2 File system consistency

For proper operation, the internal metadata of the file system and its data blocks should be in a *consistent* state. By *metadata consistency*, we mean that the state of the various metadata structures obeys a set of invariants that the file system relies on. For example, a directory entry should only point to a valid file metadata structure; if a directory points to file metadata that is uninitialized (*i.e.*, marked free), the file system is said to be *inconsistent*.

Most file systems provide metadata consistency, since that is crucial to correct operation. A stronger form of consistency is *data consistency*, where the file system guarantees that data block contents always correspond to the file metadata structures that point to them. We discuss this issue in Section 7.1. Many modern file systems such as Linux ext3 and ReiserFS provide data consistency.

3.3 File system asynchrony

An important characteristic of most modern file systems is the *asynchrony* they exhibit during updates to data and metadata. Updates are simply buffered in memory and are written to disk only after a certain delay interval, with possible reordering among those writes. While such asynchrony is crucial for performance, it complicates consistency management. Due to asynchrony, a system crash leads to a state where an arbitrary subset of updates has

been applied on disk, potentially leading to an inconsistent on-disk state. Asynchrony of updates is the principal reason for complexity in the interaction of a file system with the disk, and hence the *raison d'être* of our logic.

4 Basic entities and notations

In this section, we define the basic entities that constitute a file system in our logic, and present their notations. In the next section, we build upon these entities to present our formalism of the operation of a file system.

4.1 Basic entities

The basic entities in our model are *containers*, *pointers*, and *generations*. A file system is simply a collection of containers. Containers are linked to each other through pointers. Each file system differs in the exact types of containers it defines and the relationship it allows between those container types; we believe that this abstraction based on containers and pointers is general to describe any file system.

Containers in a file system can be *freed* and *reused*; a container is considered to be free when it is not pointed to by any other container; it is *live* otherwise. The instance of a container between a reuse and the next free is called a *generation*; thus, a generation is a specific incarnation of a container. Generations are never reused. When a container is reused, the previous generation of that container is freed and a new generation of the container comes to life. A generation is thus fully defined by its container plus a logical *generation number* that tracks how many times the container was reused. Note that generation does *not* refer to the *contents* of a container, but is an abstraction for its current incarnation; contents can change without affecting the generation.

We illustrate the notion of containers and generations with a simple example from a typical UNIX-based file system. If the file system contains a fixed set of designated *inodes*, each inode slot is a *container*. At any given point, an inode slot in use is associated with an inode *generation* that corresponds to a specific file. When the file is deleted, the corresponding inode generation is deleted (forever), but the inode container is simply marked free. A different file created later can reuse the same inode container for a logically different inode generation.

Note that a single container (*e.g.*, an inode) can point to multiple containers (*e.g.*, data blocks). A single container can also be sometimes pointed to by multiple containers (*e.g.*, hard links in UNIX file systems).

4.2 Notations

The notations used to depict the basic entities and the relationships across them are listed in Table 1. Note that many notations in the table are defined only later in the section. Containers are denoted by upper case letters, while generations are denoted by lower case letters. An “entity” in the description represents a container or a generation.

Symbol	Description
$\&A$	set of entities that point to container A
$*A$	set of entities pointed to by container A
$ A $	container that tracks if container A is live
$\&a$	set of entities that point to generation a
$*a$	set of entities pointed to by generation a
$A \rightarrow B$	denotes that container A has a pointer to B
$\&A = \emptyset$	denotes that no entity points to A
A^k	the k^{th} epoch of container A
$t(A^k)$	type of k^{th} epoch of container A
$g(A^k)$	generation of the k^{th} epoch of container A
$C(a)$	container associated with generation a
A_k	generation k of container A

Table 1: Notations on containers and generations.

A pointer is denoted by the \rightarrow symbol; $A \rightarrow B$ indicates that container A has a pointer to container B , *i.e.*, $(A \in \&B) \wedge (B \in *A)$. For most of this paper, we only consider pointers from and to containers that are live. In Section 9, we will relax this assumption and introduce a new notation for pointers involving dead containers.

4.3 Attributes of containers

To make the logic expressive for modern file systems, we extend its vocabulary with attributes on a container; a generation has the same attributes as its container.

4.3.1 Epoch

The *epoch* of a container is defined as follows: every time the *contents* of a container change *in memory*, its epoch is incremented. For example, if the file system sets different fields in an inode one after the other, each step results in a new epoch of the inode container. Since the file system can batch multiple changes to the contents due to buffering, the set of epochs visible at the disk is a subset of the total set of epochs a container goes through. We denote an epoch by the superscript notation; A^k denotes the k^{th} epoch of A . Note that our definition of epoch is only used for expressivity of our logic; it does not imply that the file system tracks such an epoch. Also note the distinction between an *epoch* and a *generation*; a generation change occurs only on a reuse of the container, while an epoch changes on every change in contents *or* when the container is reused.

4.3.2 Type

Containers can have a certain *type* associated with them. The type of a container can either be *static*, *i.e.*, it does not change during the lifetime of the file system, or can be *dynamic*, where the same container can belong to different types at different points in time. For example, in FFS-based file systems, *inode* containers are statically typed, while block containers may change their type between data, directory, and indirect pointers. We denote the type of a container A by the notation $t(A)$.

4.3.3 Shared vs. unshared

A container that is pointed to by more than one container is called a *shared container*; a container that has exactly one pointer leading into it is unshared. By default, we assume that containers are shared. We denote unshared containers with the \oplus operator. $\oplus A$ indicates that A is unshared. Note that being unshared is a property of the container *type* that the file system always ensures; a container belonging to a type that is unshared will always have only one pointer pointing into it. For example, most file systems designate data block containers to be unshared.

4.4 Memory and disk versions of containers

A file system needs to manage its structures across two domains: volatile memory and disk. Before accessing the contents of a container, the file system needs to *read* the on-disk version of the container into memory. Subsequently, the file system makes modifications to the in-memory copy of the container, and such modified contents are periodically written to disk. Thus, until the file system writes a modified container to disk, the contents of the container in memory will be different from that on disk.

5 The Formalism

We now present our formal model of the operation of a file system. We first formulate the logic in terms of *beliefs* and *actions*, and then introduce the operators in the logic, our proof system, and the basic axioms in the logic.

5.1 Beliefs

The state of the system is modeled using *beliefs*. A belief represents a certain state in memory or disk.

Any statement enclosed within $\{\}$ represents a belief. Beliefs can be either *in memory* beliefs or *on disk* beliefs, and are denoted as either $\{\}_M$ or $\{\}_D$ respectively. For example $\{A \rightarrow B\}_M$ indicates that $A \rightarrow B$ is a belief in the file system memory, *i.e.*, container A currently points to B in memory, while $\{A \rightarrow B\}_D$ means it is a disk belief. The timing of when such a belief begins to hold is determined in the context of a *formula* in our logic, as we describe in the next subsection; in brief terms, the timing of a belief is defined *relative* to other beliefs or actions specified in the formula. An isolated belief in itself thus has no temporal dimension.

While memory beliefs just represent the state the file system tracks in memory, on-disk beliefs are defined as follows: a belief holds on disk at a given time, if on a crash, the file system can conclude with the same belief purely based on a scan of on-disk state at that time. On-disk beliefs are thus solely dependent on on-disk data.

Since the file system manages free and reuse of containers, its beliefs can be in terms of *generations*; for example $\{A_k \rightarrow B_j\}_M$ is valid (note that A_k refers to generation k of container A). However, on-disk beliefs can only deal with containers, since generation information is lost at the

disk. In Sections 9 and 10, we propose techniques to expose generation information to the disk, and show that it enables improved guarantees.

5.2 Actions

The other component of our logic is *actions*, which result in changes to system state; actions thus alter the set of beliefs that hold at a given time. There are two actions defined in our logic:

- *read*(A) – This operation is used by the file system to read the contents of an on-disk container (and thus, its current generation) into memory. The file system needs to have the container in memory before it can modify it. After a *read*, the contents of A in memory and on-disk are the same, *i.e.*, $\{A\}_M = \{A\}_D$.
- *write*(A) – This operation results in flushing the current contents of a container to disk. After this operation, the contents of A in memory and on-disk are the same, *i.e.*, $\{A\}_D = \{A\}_M$.

5.3 Ordering of beliefs and actions

A fundamental aspect of the interaction of a file system with disk is the *ordering* among its actions. The ordering of actions also determines the order in which beliefs are established. To order actions and the resulting beliefs, we use the *before* (\ll) and *after* (\gg) operators. Thus, $\alpha \ll \beta$ means that α occurred before β in time. Note that by *ordering* beliefs, we are using the $\{\}$ notation as both a way of indicating the *event* of creation of the belief, and the *state* of existence of a belief. For example, the belief $\{B \rightarrow A\}_M$ represents the event where the file system assigns A as one of the pointers from B .

We also use a special ordering operator called *precedes* (\prec). Only a belief can appear to the left of a \prec operator. The \prec operator is defined as follows: $\alpha \prec \beta$ means that belief α occurs before β (*i.e.*, $\alpha \prec \beta \Rightarrow \alpha \ll \beta$); further, it means that belief α holds at least until β occurs. This implies there is no intermediate action or event between α and β that invalidates belief α .

The operator \prec is *not* transitive; $\alpha \prec \beta \prec \gamma$ does not imply $\alpha \prec \gamma$, because belief α needs to hold only until β and not necessarily until γ (note that $\alpha \prec \beta \prec \gamma$ is simply a shortcut for $(\alpha \prec \beta) \wedge (\beta \prec \gamma)$ (note that this implies $\alpha \ll \gamma$).

Beliefs can be grouped using parentheses, which has the following semantics with precedes:

$$(\alpha \prec \beta) \prec \gamma \Rightarrow (\alpha \prec \beta) \wedge (\alpha \prec \gamma) \wedge (\beta \prec \gamma) \quad (1)$$

If a group of beliefs precedes a certain other belief α , *every* belief within the parentheses precedes belief α .

5.4 Proof system

Given our primitives for sequencing beliefs and actions, we can define *rules* or *formulas* in our logic in terms of

an *implication* of one event sequence given another sequence. We use the traditional operators: \Rightarrow (implication) and \Leftrightarrow (double implication, *i.e.*, if and only if). We also use logical AND (\wedge) and OR (\vee) to combine sequences.

An example of a logical rule is: $\alpha \ll \beta \Rightarrow \gamma$. This notation means that *every time* an event or action β occurs after α , event γ occurs *at the point of occurrence of* β . The rule does not say anything about *when* α or β occurs in absolute time; all it says is whenever they occur in that order, γ occurs. Thus, the above rule would be valid if $\alpha \ll \beta$ never occurred at all. In general, if the left hand side of the rule involves a more complex expression, say a disjunction of two components, the belief on the RHS holds at the point of occurrence of the first event that makes the LHS true; in the example above, the occurrence of β makes the sequence $\alpha \ll \beta$ true.

Another example of a rule is $\alpha \ll \beta \Rightarrow \alpha \ll \gamma \ll \beta$; this rule denotes that every time β occurs after α , γ should have occurred sometime between α and β . Note that in such a rule where the same event occurs in both sides, the event constitutes a temporal reference point by referring to the same time instant in both the LHS and RHS. This temporal interpretation of identical events is crucial to the above rule serving the intended implication; otherwise the RHS could refer to some other instant where $\alpha \ll \beta$.

Rules such as the above can be used in logical proofs by *event sequence substitution*; for example, with the rule $\alpha \ll \beta \Rightarrow \gamma$, whenever the subsequence $\alpha \ll \beta$ occurs in a sequence of events, it logically implies the event γ . We could then apply the above rule to any event sequence by replacing any subsequence that matches the left half of the rule, with the right half; thus, with the above rule, we have the following postulate: $\alpha \ll \beta \ll \delta \Rightarrow \gamma \ll \delta$. Thus, our proof system enables deriving new invariants about the file system, building on basic axioms.

5.5 Basic axioms

In this subsection, we present the axioms that govern the transition of beliefs across memory and disk.

- If a container B points to A in memory, its current generation also points to A in memory.

$$\{B^x \rightarrow A\}_M \Leftrightarrow \{g(B^x) \rightarrow A\}_M \quad (2)$$

- If B points to A in memory, a *write* of B will lead to the disk belief that B points to A .

$$\{B \rightarrow A\}_M \prec \text{write}(B) \Rightarrow \{B \rightarrow A\}_D \quad (3)$$

The converse states that the disk belief implies that the same belief first occurred in memory.

$$\{B \rightarrow A\}_D \Rightarrow \{B \rightarrow A\}_M \ll \{B \rightarrow A\}_D \quad (4)$$

- Similarly, if B points to A on disk, a *read* of B will result in the file system inheriting the same belief.

$$\{B \rightarrow A\}_D \prec \text{read}(B) \Rightarrow \{B \rightarrow A\}_M \quad (5)$$

- If the on-disk contents of container A pertain to epoch y , some generation c should have pointed to generation $g(A^y)$ in memory, followed by $\text{write}(A)$. The converse also holds:

$$\{A^y\}_D \Rightarrow \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \{A^y\}_D \quad (6)$$

$$\{c \rightarrow A_k\}_M \prec \text{write}(A) \Rightarrow \{A^y\}_D \wedge (g(A^y) = k) \quad (7)$$

Note that A_k refers to some generation k of A , and is used in the above rule to indicate that the generation c points to is the same as that of A^y .

- If $\{b \rightarrow A_k\}$ and $\{c \rightarrow A_j\}$ hold in memory at two different points in time, container A should have been freed between those instants.

$$\begin{aligned} &\{b \rightarrow A_k\}_M \ll \{c \rightarrow A_j\}_M \wedge (k \neq j) \\ &\Rightarrow \{b \rightarrow A_k\}_M \ll \{\&A = \emptyset\}_M \prec \{c \rightarrow A_j\}_M \end{aligned} \quad (8)$$

Note that the rule includes the scenario where an intermediate generation A_l occurs between A_k and A_j .

- If container B pointed to A on disk, and subsequently the file system removes the pointer from B to A in memory, a write of B will lead to the disk belief that B does not point to A .

$$\begin{aligned} &\{B \rightarrow A\}_D \prec \{A \notin *B\}_M \prec \text{write}(B) \\ &\Rightarrow \{A \notin *B\}_D \end{aligned} \quad (9)$$

Further, if A is an unshared container, the write of B will lead to the disk belief that no container points to A , i.e., A is *free*.

$$\begin{aligned} &\oplus A \wedge (\{B \rightarrow A\}_D \prec \{\&A = \emptyset\}_M \prec \text{write}(B)) \\ &\Rightarrow \{\&A = \emptyset\}_D \end{aligned} \quad (10)$$

- If A is a dynamically typed container, and its type at two instants are different, A should have been freed in between.

$$\begin{aligned} &(\{t(A) = x\}_M \ll \{t(A) = y\}_M) \wedge (x \neq y) \\ &\Rightarrow \{t(A) = x\}_M \ll \{\&A = \emptyset\}_M \prec \{t(A) = y\}_M \end{aligned} \quad (11)$$

5.6 Completeness of notations

The various notations we have discussed in this section cover a wide range of the set of behaviors that we would want to model in a file system. However, this is by no means a *complete* set of notations that can model every aspect of a file system. As we show in Section 7.2 and Section 9, certain specific file system features may require new notations. The main contribution of this paper lies in putting forth a framework to formally reason about file

system correctness. Although new notations may sometimes need to be introduced for certain specific file system features, much of the framework will apply without any modification.

5.7 Connections to Temporal Logic

Our logic bears some similarity to linear temporal logic. The syntax of *Linear Temporal Logic (LTL)* [5, 15] is defined as follows:

- A formula $p \in AP$ is an LTL formula, where AP is a set of atomic propositions.
- Given two LTL formulas f and g , $\neg f$, $f \wedge g$, $f \vee g$, $\mathbf{X} f$, $\mathbf{F} f$, $\mathbf{G} f$, $f \mathbf{U} g$, and $f \mathbf{R} g$ are LTL formulas.

In the definition given above \mathbf{X} (“next time”), \mathbf{F} (“in the future”), \mathbf{G} (“always”), \mathbf{U} (“until”), and \mathbf{R} (“release”) are temporal operators. Our formalism is a fragment of LTL, where the set of atomic propositions AP consists of memory and disk beliefs and actions and only temporal operators \mathbf{F} and \mathbf{U} are allowed. In our formalism, $\alpha \ll \beta$ and $\alpha \prec \beta$ are equivalent to $\alpha \mathbf{F} \beta$ and $\alpha \mathbf{U} \beta$, respectively.

Given an execution π , which is a sequence of states, and an LTL formula f , $\pi \models f$ denotes that f is true in the execution π . A system S satisfies an LTL formula f if all its executions satisfy f . The precise semantics of the satisfaction relation (the meaning of \models) can be found in [5, Chapter 3]. Thus the semantics for our formalism follows from the standard semantics of LTL.

In our proof system, we are given set of axioms \mathcal{A} (given in Section 5.5) and a desired property f (such as the data consistency property described in Section 7.1), and we want to prove that f follows from the axioms in \mathcal{A} (denoted by $\mathcal{A} \rightarrow f$), i.e., if a file system satisfies all properties in the set \mathcal{A} , it will also satisfy property f .

6 File System Properties

Various file systems provide different guarantees on their update behavior. Each guarantee translates into new rules to the logical model of the file system, and can be used to complement our basic rules when reasoning about that file system. In this section, we discuss three such properties.

6.1 Container exclusivity

A file system exhibits *container exclusivity* if it guarantees that for every on-disk container, there is at most one dirty copy of the container’s contents in the file system cache. It also requires the file system to ensure that the in-memory contents of a container do not change while the container is being written to disk. Many file systems such as BSD FFS, Linux ext2 and VFAT exhibit container exclusivity; some journaling file systems like ext3 do not exhibit this property. In our equations, when we refer to containers in memory, we refer to the latest epoch of the container in memory, in the case of file systems that do not obey container exclusivity. For example, in eq. 10, $\{\&A = \emptyset\}_M$ means that at that time, there is no container whose latest

epoch in memory points to A ; similarly, $write(B)$ means that the latest epoch of B at that time is being written. When referring to a specific version, we use the epoch notation. Of course, if container exclusivity holds, only one epoch of any container exists in memory.

Under container exclusivity, we have a stronger converse for eq. 3:

$$\{B \rightarrow A\}_D \Rightarrow \{B \rightarrow A\}_M \prec \{B \rightarrow A\}_D \quad (12)$$

If we assume that A is unshared, we have a stronger equation following from equation 12, because the only way the disk belief $\{B \rightarrow A\}_D$ can hold is if B was written by the file system. Note that many containers in typical file systems (such as data blocks) are unshared.

$$\{B \rightarrow A\}_D \Rightarrow \{B \rightarrow A\}_M \prec (write(B) \ll \{B \rightarrow A\}_D) \quad (13)$$

6.2 Reuse ordering

A file system exhibits *reuse ordering* if it ensures that before reusing a container, it commits the freed state of the container to disk. For example, if A is pointed to by generation b in memory, later freed (*i.e.*, $\&A = \emptyset$), and then another generation c is made to point to A , the freed state of A (*i.e.*, the container of generation b , with its pointer removed) is written to disk before the reuse occurs.

$$\{b \rightarrow A\}_M \prec \{\&A = \emptyset\}_M \prec \{c \rightarrow A\}_M \\ \Rightarrow \{\&A = \emptyset\}_M \prec write(C(b)) \ll \{c \rightarrow A\}_M$$

Since every reuse results in such a commit of the freed state, we could extend the above rule as follows:

$$\{b \rightarrow A\}_M \ll \{\&A = \emptyset\}_M \prec \{c \rightarrow A\}_M \\ \Rightarrow \{\&A = \emptyset\}_M \prec write(C(b)) \ll \{c \rightarrow A\}_M \quad (14)$$

FFS with soft updates [6] and Linux ext3 are two examples of file systems that exhibit reuse ordering.

6.3 Pointer ordering

A file system exhibits *pointer ordering* if it ensures that before writing a container B to disk, the file system writes all containers that are pointed to by B .

$$\{B \rightarrow A\}_M \prec write(B) \\ \Rightarrow \{B \rightarrow A\}_M \prec (write(A) \ll write(B)) \quad (15)$$

FFS with soft updates is an example of a file system that exhibits pointer ordering.

7 Modeling Existing Systems

Having defined the basic formalism of our logic, we proceed to using the logic to model and reason about file system behaviors. In this section, we present proofs for two properties important for file system consistency. First, we discuss the *data consistency* problem in a file system. We then model a journaling file system and reason about the *non-rollback* property in a journaling file system.

7.1 Data consistency

We first consider the problem of *data consistency* of the file system after a crash. By data consistency, we mean that the contents of data block containers have to be consistent with the metadata that references the data blocks. In other words, a file should not end up with data from a different file when the file system recovers after a crash. Let us assume that B is a file metadata container (*i.e.* contains pointers to the data blocks of the respective file), and A is a data block container. Then, if the disk belief that B^x points to A holds, and the on-disk contents of A were written when k was the generation of A , then epoch B^x should have pointed (at some time in the past) exactly to the k^{th} generation of A in memory, and not a different generation. The following rule summarizes this:

$$\{B^x \rightarrow A\}_D \wedge \{A^y\}_D \Rightarrow (\{B^x \rightarrow A_k\}_M \ll \{B^x \rightarrow A\}_D) \\ \wedge (k = g(A^y))$$

We prove below that if the file system exhibits reuse ordering and pointer ordering, it never suffers a data consistency violation. We also show that if the file system does not obey any such ordering, data consistency could be compromised on crashes.

For simplicity, let us make a further assumption that the data containers in our file system are nonshared ($\oplus A$), *i.e.*, different files do not share data block pointers. Let us also assume that the file system obeys the container exclusivity property. Many modern file systems such as ext2 and VFAT have these properties. Since under block exclusivity $\{B^x \rightarrow A\}_D \Rightarrow \{B^x \rightarrow A\}_M \prec \{B^x \rightarrow A\}_D$ (by eq. 12), we can rewrite the above rule as follows:

$$(\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D) \wedge \{A^y\}_D \\ \Rightarrow (k = g(A^y)) \quad (16)$$

If this rule does not hold, it means that the file represented by the generation $g(B^x)$ points to a generation k of A , but the contents of A were written when its generation was $g(A^y)$, clearly a case of data corruption.

To show that this rule does not always hold, we assume the negation and prove that it is reachable as a sequence of valid file system actions ($\alpha \Rightarrow \beta \equiv \neg(\alpha \wedge \neg\beta)$).

From eq. 6, we have $\{A^y\}_D \Rightarrow \{c \rightarrow g(A^y)\}_M \prec write(A)$. Thus, we have two event sequences implied by the LHS of eq. 16:

- i. $\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D$
- ii. $\{c \rightarrow g(A^y)\}_M \prec write(A)$

Thus, in order to prove eq. 16, we need to prove that every possible interleaving of the above two sequences, together with the clause $(k \neq g(A^y))$ is invalid. To disprove eq. 16, we need to prove that at least one of the interleavings is valid.

Since $(k \neq g(A^y))$, and since $\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow$

$A\}_D$, the event $\{c \rightarrow g(A^y)\}_M$ cannot occur in between those two events, due to container exclusivity and because A is unshared. Similarly $\{B^x \rightarrow A_k\}_M$ cannot occur between $\{c \rightarrow g(A^y)\}_M \prec \text{write}(A)$. Thus, we have only two interleavings:

1. $\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \ll \{c \rightarrow g(A^y)\}_M \prec \text{write}(A)$
2. $\{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D$

Case 1:

Applying eq. 2,

$$\Rightarrow \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \\ \ll \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \wedge (k \neq g(A^y))$$

Applying eq. 8,

$$\Rightarrow \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \\ \ll \{\&A = \emptyset\}_M \prec \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \quad (17)$$

Since step 17 is a valid sequence in file system execution, where generation A^k could be freed due to a delete of the file represented by generation $g(B^x)$ and then a subsequent generation of the block is reallocated to the file represented by generation c in memory, we have shown that this violation could occur.

Let us now assume that our file system obeys reuse ordering, *i.e.*, equation 14. Under this additional constraint, equation 17 would imply the following:

$$\Rightarrow \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \prec \\ \{\&A = \emptyset\}_M \prec \text{write}(B) \ll \\ \{c \rightarrow g(A^y)\}_M \prec \text{write}(A)$$

By eq. 10,

$$\Rightarrow \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \prec \\ \{\&A = \emptyset\}_D \ll \{c \rightarrow g(A^y)\}_M \prec \\ \text{write}(A) \\ \Rightarrow \{\&A = \emptyset\}_D \wedge \{A_c\}_D \quad (18)$$

This is however, a contradiction under the initial assumption we started off with, *i.e.* $\{\&A = B\}_D$. Hence, under reuse ordering, we have shown that this particular scenario does not arise at all.

Case 2: $\{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \wedge (k \neq g(A^y))$

Again, applying eq. 2,

$$\Rightarrow (k \neq g(A^y)) \wedge \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \\ \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D$$

By eqn 8,

$$\Rightarrow \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \{\&A = \emptyset\}_M \\ \prec \{g(B^x) \rightarrow A_k\}_M \prec \{B^x \rightarrow A\}_D \quad (19)$$

Again, this is a valid file system sequence where file generation c pointed to data block generation $g(A^y)$, the generation $g(A^y)$ gets deleted, and a new generation k of

container A gets assigned to file generation $g(B^x)$. Thus, consistency violation can also occur in this scenario.

Interestingly, when we apply eq. 14 here, we get

$$\Rightarrow \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \{\&A = \emptyset\}_M \\ \prec \text{write}(C(c)) \ll \{g(B^x) \rightarrow A_k\}_M \\ \prec \{B^x \rightarrow A\}_D$$

However, we cannot apply eq. 10 in this case because the belief $\{C \rightarrow A\}_D$ need not hold. Even if we did have a rule that led to the belief $\{\&A = \emptyset\}_D$ immediately after $\text{write}(C(c))$, that belief will be overwritten by $\{B^x \rightarrow A\}_D$ later in the sequence. Thus, eq. 14 does not invalidate this sequence; reuse ordering thus does not guarantee data consistency in this case.

Let us now make another assumption, that the file system also obeys pointer ordering (eq. 15).

Since we assume that A is unshared, and that container exclusivity holds, we can apply eq. 13 to equation 19.

$$\Rightarrow \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \{\&A = \emptyset\}_M \prec \\ \{g(B^x) \rightarrow A_k\}_M \prec \text{write}(B) \ll \{B^x \rightarrow A\}_D \quad (20)$$

Now applying the pointer ordering rule (eqn 15.),

$$\Rightarrow \{c \rightarrow g(A^y)\}_M \prec \text{write}(A) \ll \{\&A = \emptyset\}_M \prec \\ \{g(B^x) \rightarrow A_k\}_M \prec \text{write}(A) \ll \text{write}(B) \\ \ll \{B^x \rightarrow A\}_D$$

By eq. 7,

$$\Rightarrow \{c \rightarrow A\}_M \prec \text{write}(A) \ll \{\&A = \emptyset\}_M \prec \\ \{A^y\}_D \ll \text{write}(B) \ll \{B^x \rightarrow A\}_D \wedge (k = g(A^y)) \\ \Rightarrow \{A^y\}_D \wedge \{B^x \rightarrow A\}_D \wedge (k = g(A^y)) \quad (21)$$

This is again a contradiction, since this implies that the contents of A on disk belong to the same generation A_k , while we started out with the assumption that $g(A^y) \neq k$.

Thus, under reuse ordering and pointer ordering, the file system never suffers a data consistency violation. If the file system does not obey any such ordering (such as ext2), data consistency could be compromised on crashes. Note that this inconsistency is fundamental, and cannot be fixed by scan-based consistency tools such as *fsck*. We also verified that this inconsistency occurs in practice; we were able to reproduce this case experimentally on an ext2 file system.

7.2 Modeling file system journaling

We now extend our logic with rules that define the behavior of a journaling file system. We then use the model to reason about a key property in a journaling file system.

Journaling is a technique commonly used by file systems to ensure metadata consistency. When a single file system operation spans multiple changes to metadata structures, the file system groups those changes into a *transaction* and guarantees that the transaction commits atomically, thus preserving consistency. To provide atomicity, the file system first writes the changes to a *write-*

ahead log (WAL), and propagates the changes to the actual on-disk location only after the transaction is *committed* to the log. A transaction is committed when all changes are logged, and a special “commit” record is written to log indicating completion of the transaction. When the file system recovers after a crash, a checkpointing process *replays* all changes that belong to committed transactions.

To model journaling, we consider a logical “transaction” object that determines the set of *log record* containers that belong to that transaction, and thus logically contains pointers to the log copies of all containers modified in that transaction. We denote the log copy of a journaled container by the $\hat{\cdot}$ symbol on top of the container name; \hat{A}^x is thus a container in the *log*, i.e., *journal* of the file system (note that we assume *physical logging*, such as the block-level logging in ext3). The physical realization of the transaction object is the “commit” record, since it logically points to all containers that changed in that transaction. For the WAL property to hold, the commit container should be written only after the log copy of all modified containers that the transaction points to are written.

If T is the commit container, the WAL property leads to the following two rules:

$$\{T \rightarrow \hat{A}^x\}_M \prec \text{write}(T) \Rightarrow \{T \rightarrow \hat{A}^x\}_M \prec (\text{write}(\hat{A}^x) \ll \text{write}(T)) \quad (22)$$

$$\{T \rightarrow \hat{A}^x\}_M \prec \text{write}(A^x) \Rightarrow \{T \rightarrow \hat{A}^x\}_M \prec (\text{write}(T) \ll \text{write}(A^x)) \quad (23)$$

The first rule states that the transaction is not committed (i.e., commit record not written) until all containers belonging to the transaction are written to disk. The second rule states that the on-disk home copy of a container is written only after the transaction in which the container was modified, is committed to disk. Note that unlike the normal pointers considered so far that point to containers or generations, the pointers from container T in the above two rules point to *epochs*. These *epoch pointers* are used because a commit record is associated with a specific epoch (e.g., snapshot) of the container.

The replay or checkpointing process can be depicted by the following two rules.

$$\{T \rightarrow \hat{A}^x\}_D \wedge \{T\}_D \Rightarrow \text{write}(A^x) \ll \{A^x\}_D \quad (24)$$

$$\{T_1 \rightarrow \hat{A}^x\}_D \wedge \{T_2 \rightarrow \hat{A}^y\}_D \wedge (\{T_1\}_D \ll \{T_2\}_D) \Rightarrow \text{write}(A^y) \ll \{A^y\}_D \quad (25)$$

The first rule says that if a container is part of a transaction and the transaction is committed on disk, the on-disk copy of the container is updated with the logged copy pertaining to that transaction. The second rule says that if the same container is part of multiple committed transactions, the on-disk copy of the container is updated with the copy pertaining to the last of those transactions.

The following belief transitions hold:

$$(\{T \rightarrow \hat{B}^x\}_M \wedge \{B^x \rightarrow A\}_M) \prec \text{write}(T)$$

$$\Rightarrow \{B^x \rightarrow A\}_D \quad (26)$$

$$\{T \rightarrow \hat{A}^x\}_M \prec \text{write}(T) \Rightarrow \{A^x\}_D \quad (27)$$

Rule 26 states that if B^x points to A and \hat{B}^x belongs to transaction T , the commit of T leads to the disk belief $\{B^x \rightarrow A\}_D$. Rule 27 says that the disk belief $\{A^x\}_D$ holds immediately on commit of the transaction which \hat{A}^x is part of; creation of the belief does not require the checkpoint write to happen. As described in §5.1, a disk belief pertains to the belief the file system would reach, if it were to start from the current disk state.

In certain journaling file systems, it is possible that only containers of certain types are journaled; updates to other containers directly go to disk, without going through the transaction machinery. In our proofs, we will consider the cases of both complete journaling (where all containers are journaled) and selective journaling (only containers of a certain type). In the selective case, we also address the possibility of a container changing its type from a journaled type to a non-journaled type and vice versa. For a container B that belongs to a journaling type, we have the following converse of equation 26:

$$\{B^x \rightarrow A\}_D \Rightarrow (\{T \rightarrow \hat{B}^x\}_M \wedge \{B^x \rightarrow A\}_M) \prec \text{write}(T) \ll \{B^x \rightarrow A\}_D \quad (28)$$

We can show that in complete journaling, data inconsistency never occurs; we omit this due to space constraints.

7.2.1 The non-rollback property

We now introduce a new property called *non-rollback* that is pertinent to file system consistency. We first formally define the property and then reason about the conditions required for it to hold in a journaling file system.

The non-rollback property states that the contents of a container on disk are never overwritten by *older* contents from a previous epoch. This property can be expressed as:

$$\{A^x\}_D \ll \{A^y\}_D \Rightarrow \{A^x\}_M \ll \{A^y\}_M \quad (29)$$

The above rule states that if the on-disk contents of A move from epoch x to y , it should logically imply that epoch x occurred before epoch y in memory as well. The non-rollback property is crucial in journaling file systems; absence of the property could lead to data corruption.

In the proof below, we logically derive the corner cases that need to be handled for this property to hold, and show that *journal revoke records* effectively ensure this.

If the disk believes in the x^{th} epoch of A , there are only two possibilities. If the type of A^x was a journaled type, A^x should have belonged to a transaction and the disk must have observed the commit record for the transaction; as indicated in eq 27, the belief of $\{A^x\}_D$ occurs immediately after the commit. However, at a later point the actual contents of A^x will be written by the file system as part of its checkpoint propagation to the actual on-disk

location, thus re-establishing belief $\{A^x\}_D$. If J is the set of all journaled types,

$$\begin{aligned} \{A^x\}_D \wedge \{t(A^x) \in J\}_M &\Rightarrow (\{A^x\}_M \wedge \{T \rightarrow \hat{A}^x\}_M) \\ &\prec \text{write}(T) \ll \{A^x\}_D \\ &\ll \text{write}(A^x) \ll \{A^x\}_D \quad (30) \end{aligned}$$

The second possibility is that A^x is of a type that is not journaled. In this case, the only way the disk could have learnt of it is by a prior commit of A^x .

$$\begin{aligned} \{A^x\}_D \wedge \{t(A^x) \notin J\}_M &\Rightarrow \{A^x\}_M \prec \text{write}(A^x) \\ &\ll \{A^x\}_D \quad (31) \end{aligned}$$

A^x and A^y are journaled:

Let us first assume that both A^x and A^y belong to a journaled type. To prove the non-rollback property, we consider the LHS of eq. 29: $\{A^x\}_D \ll \{A^y\}_D$; since both A^x and A^y are journaled, we have the following two sequence of events that led to the two beliefs (by eq. 30):

$$\begin{aligned} (\{A^x\}_M \wedge \{T_1 \rightarrow \hat{A}^x\}_M) &\prec \text{write}(T_1) \ll \{A^x\}_D \\ &\ll \text{write}(A^x) \ll \{A^x\}_D \end{aligned}$$

$$\begin{aligned} (\{A^y\}_M \wedge \{T_2 \rightarrow \hat{A}^y\}_M) &\prec \text{write}(T_2) \ll \{A^y\}_D \\ &\ll \text{write}(A^y) \ll \{A^y\}_D \end{aligned}$$

Omitting the *write* actions in the above sequences for simplicity, we have the following sequences of events:

- i. $\{A^x\}_M \ll \{A^x\}_D \ll \{A^x\}_D$
- ii. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D$

Note that in each sequence, there are two instances of the *same* disk belief being created: the first instance is created when the corresponding transaction is committed, and the second instance when the checkpoint propagation happens at a later time. In snapshot-based coarse-grained journaling systems (such as ext3), transactions are always committed in order. Thus, if epoch A^x occurred before A^y , T_1 will be committed before T_2 (*i.e.*, the first instance of $\{A^x\}_D$ will occur before the first instance of $\{A^y\}_D$). Another property true of such journaling is that the checkpointing is in-order as well; if there are two committed transactions with different copies of the same data, only the version pertaining to the later transaction is propagated during the checkpoint.

Thus, the above two sequences of events lead to only two interleavings, depending on whether epoch x occurs before epoch y or vice versa. Once the ordering between epoch x and y is fixed, the rest of the events are constrained to a single sequence:

Interleaving 1:

$$\begin{aligned} &(\{A^x\}_M \ll \{A^y\}_M) \wedge (\{A^x\}_D \ll \{A^y\}_D \ll \{A^y\}_D) \\ \Rightarrow &\{A^x\}_M \ll \{A^y\}_M \end{aligned}$$

Interleaving 2:

$$\begin{aligned} \Rightarrow &(\{A^y\}_M \ll \{A^x\}_M) \wedge (\{A^y\}_D \ll \{A^x\}_D \ll \{A^x\}_D) \\ \Rightarrow &\{A^y\}_D \ll \{A^x\}_D \end{aligned}$$

Thus, the second interleaving results in a contradiction from our initial statement we started with (*i.e.*, $\{A^x\}_D \ll \{A^y\}_D$). Therefore the first interleaving is the only legal way the two sequences of events could be combined. Since the first interleaving implies that $\{A^x\}_M \ll \{A^y\}_M$, we have proved that if the two epochs are journaled, the non-rollback property holds.

A^y is journaled, but A^x is not:

We now consider the case where the type of A changes between epochs x and y , such that A^y belongs to a journaled type and A^x does not.

We again start with the statement $\{A^x\}_D \ll \{A^y\}_D$. From equations 30 and 31, we have the following two sequences of events:

- i. $(\{A^y\}_M \wedge \{T \rightarrow \hat{A}^y\}_M) \prec \text{write}(T)$
 $\ll \{A^y\}_D \ll \text{write}(A^y) \ll \{A^y\}_D$
- ii. $\{A^x\}_M \prec \text{write}(A^x) \ll \{A^x\}_D$

Omitting the *write* actions for the sake of readability, the sequences become:

- i. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D$
- ii. $\{A^x\}_M \ll \{A^x\}_D$

To prove the non-rollback property, we need to show that every possible interleaving of the above two sequences where $\{A^y\}_M \ll \{A^x\}_M$ results in a contradiction, *i.e.*, cannot co-exist with $\{A^x\}_D \ll \{A^y\}_D$.

The interleavings where $\{A^y\}_M \ll \{A^x\}_M$ are:

1. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^x\}_D \ll \{A^y\}_D \ll \{A^y\}_D$
2. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^x\}_D \ll \{A^y\}_D$
3. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^x\}_D$
4. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^x\}_D \ll \{A^y\}_D$
5. $\{A^y\}_M \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^y\}_D \ll \{A^x\}_D$
6. $\{A^y\}_M \ll \{A^y\}_D \ll \{A^x\}_M \ll \{A^y\}_D \ll \{A^x\}_D$

Scenarios 3, 5, and 6 imply $\{A^y\}_D \ll \{A^x\}_D$ and are therefore invalid interleavings. Scenarios 1, 2, and 4 are valid interleavings that do not contradict our initial assumption of disk beliefs, but at the same time, imply $\{A^y\}_M \ll \{A^x\}_M$; these scenarios thus violate the non-rollback property. Therefore, under dynamic typing, the above journaling mechanism does not guarantee non-rollback. Due to this violation, file contents can be corrupted by stale metadata generations.

Scenario 2 and 4 occur because the checkpoint propagation of earlier epoch A^y which was journaled, occurs *after* A was overwritten as a non-journaled epoch. To prevent this, we need to impose that the checkpoint propagation of a container in the context of transaction T does not

happen if the on-disk contents of that container were updated *after* the commit of T . The *journal revoke records* in ext3 precisely guarantee this; if a revoke record is encountered during log replay (during a pre-scan of the log), the corresponding block is not propagated to the actual disk location.

Scenario 1 happens because a later epoch of A is committed to disk before the transaction which modified an earlier epoch is committed. To prevent this, we need a form of *reuse ordering*, which imposes that before a container changes type (i.e. is reused in memory), the transaction that freed the previous generation be committed. Since transactions commit in order, and the freeing transaction should occur *after* transaction T which used A^y in the above example, we have the following guarantee:

$$\begin{aligned} \{t(A^y) \in J\}_M \wedge \{t(A^x) \notin J\}_M \wedge (\{A^y\}_M \ll \{A^x\}_M) \\ \Rightarrow \{A^y\}_M \prec write(T) \ll \{A^x\}_M \end{aligned}$$

With this rule, Scenario 1 becomes the same as 2 and 4 and is handled by the revoke record solution. Thus, under these two properties, the non-rollback property holds.

8 Redundant Synchrony in Ext3

We examine a performance problem with the ext3 file system where the transaction commit procedure artificially limits parallelism due to a redundant synchrony in its disk writes [16]. The *ordered mode* of ext3 guarantees that a newly created file will never point to stale data blocks after a crash. Ext3 ensures this guarantee by the following ordering in its commit procedure: when a transaction is committed, ext3 first writes to disk the data blocks allocated in that transaction, waits for those writes to complete, then writes the journal blocks to disk, waits for those to complete, and then writes the commit block. If I is an inode container, F is a file data block container, and T is the transaction commit container, the commit procedure of ext3 can be expressed by the following equation:

$$\begin{aligned} (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I}^x\}_M) \prec write(T) \\ \Rightarrow (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I}^x\}_M) \\ \prec write(F) \ll write(\hat{I}^x) \ll write(T) \end{aligned} \quad (32)$$

To examine if this is a necessary condition to ensure the no-stale-data guarantee, we first formally depict the guarantee that the ext3 ordered mode seeks to provide, in the following equation:

$$\begin{aligned} \{I^x \rightarrow F_k\}_M \ll \{I^x \rightarrow F\}_D \Rightarrow \{F_y\}_D \ll \{I^x \rightarrow F\}_D \\ \wedge (g(F^y) = k) \end{aligned} \quad (33)$$

The above equation states that if the disk acquires the belief that $\{I^x \rightarrow F\}$, then the contents of the data container F on disk should *already* pertain to the generation of F that I^x pointed to in memory. Note that because ext3 obeys reuse ordering, the ordered mode guarantee only

needs to cater to the case of a *free* data block container being allocated to a new file.

We now prove equation 33, examining the conditions that need to hold for this equation to be true. We consider the LHS of the equation:

$$\{I^x \rightarrow F_k\}_M \ll \{I^x \rightarrow F\}_D$$

Applying equation 28 to the above, we get

$$\begin{aligned} \Rightarrow (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I}^x\}_M) \prec \\ write(T) \ll \{I^x \rightarrow F\}_D \end{aligned}$$

Applying equation 32, we get

$$\begin{aligned} \Rightarrow (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I}^x\}_M) \prec \\ write(F) \ll write(\hat{I}^x) \ll \\ write(T) \ll \{I^x \rightarrow F\}_D \end{aligned} \quad (34)$$

By equation 7,

$$\begin{aligned} \Rightarrow (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I}^x\}_M) \prec \\ \{F^y\}_D \ll write(\hat{I}^x) \ll \\ write(T) \ll \{I^x \rightarrow F\}_D \wedge (g(F^y) = k) \\ \Rightarrow \{F^y\}_D \ll \{I^x \rightarrow F\}_D \wedge (g(F^y) = k) \end{aligned}$$

Thus, the current ext3 commit procedure (equation 32) guarantees the no-stale-data property. However, to see if all the waits in the above procedure are required, let us reorder the two actions $write(F)$ and $write(\hat{I}^x)$ in eq. 34:

$$\begin{aligned} \Rightarrow (\{I^x \rightarrow F_k\}_M \wedge \{T \rightarrow \hat{I}^x\}_M) \prec \\ write(\hat{I}^x) \ll write(F) \ll \\ write(T) \ll \{I^x \rightarrow F\}_D \end{aligned}$$

Once again, applying equation 7,

$$\Rightarrow \{F^y\}_D \ll \{I^x \rightarrow F\}_D \wedge (g(F^y) = k)$$

Thus, we can see that the ordering between the actions $write(F)$ and $write(\hat{I}^x)$ is inconsequential to the guarantee that ext3 ordered mode attempts to provide. We can hence conclude that the wait that ext3 employs after the write to data blocks is redundant, and unnecessarily limits its parallelism between data and journal writes. This can have especially severe performance implications in settings where the log is stored on a separate disk, as illustrated in previous work [16].

We believe that this specific example points to a general problem with file system design. When developers do not have rigorous frameworks to reason about correctness, they tend to be *conservative*. Such conservatism often translates into unexploited opportunities for performance optimization. A systematic framework enables aggressive optimizations while ensuring correctness.

9 Support for Consistent Undelete

In this section, we demonstrate that our logic enables one to quickly formulate and prove properties about new file

system features and mechanisms. We explore a functionality that is traditionally not considered a part of core file system design: the ability to *undelete* deleted files with certain consistency guarantees. The ability to recover deleted files is useful, as demonstrated by the large number of tools available for the purpose [17, 19]. Such tools try to rebuild deleted files by scavenging through on-disk metadata; this is possible to an extent because file systems do not normally zero out freed metadata containers (they are simply marked free). For example, in a UNIX file system, the block pointers in a deleted inode would indicate the blocks that used to belong to that deleted file.

However, none of the existing tools for undelete can guarantee *consistency* (i.e., assert that the recovered contents are valid). While undelete is fundamentally only best-effort (files cannot be recovered if the blocks were subsequently reused in another file), the user needs to know how trustworthy the recovered contents are. We demonstrate using our logic that with existing file systems, such *consistent* undelete is impossible. We then provide a simple solution, and prove that the solution guarantees consistent undelete. Finally, we present an implementation of the solution in ext3.

9.1 Undelete in existing systems

To model undelete, the logic needs to express pointers from containers holding a *dead* generation. We introduce the \rightsquigarrow notation to indicate such a pointer, which we call a *dead pointer*. We also define a new operator $\&$ on a container that denotes the set of all dead *and* live entities pointing to the container. Let $undel(B)$ be the undelete action on container B . The undelete process can be summarized by the following equation:

$$\begin{aligned} undel(B) \wedge \{B^x \rightsquigarrow A\}_D \wedge \{\&A = \{B\}\}_D \\ \Leftrightarrow \{B^x \rightsquigarrow A\}_D \prec \{B^y \rightarrow A\}_D \wedge (g(B^x) = g(B^y)) \end{aligned} \quad (35)$$

In other words, if the dead (free) container B^x points to A on disk, and is the only container (alive or dead) pointing to A , the undelete makes the generation $g(B^x)$ live again, and makes it point to A .

The guarantee we want to hold for consistency is that if a dead pointer from B^x to A is brought alive, the on-disk contents of A at the time the pointer is brought alive must correspond to the same generation that epoch B^x originally pointed to in memory (similar to the data consistency formulation in §7.1):

$$\begin{aligned} \{B^x \rightarrow A\}_M \ll \{B^x \rightsquigarrow A\}_D \prec \{B^y \rightarrow A\}_D \\ \wedge (g(B^x) = g(B^y)) \\ \Rightarrow \{B^x \rightsquigarrow A\}_D \wedge \{A^z\}_D \wedge (g(A^z) = k) \end{aligned}$$

Note that the clause $g(B^x) = g(B^y)$ is required in the LHS to cover only the case where the *same* generation is brought to life, which would be true only for undelete.

To show that the above guarantee does not hold necessarily, we consider the negation of the RHS, i.e., $\{A^z\}_D \wedge (g(A^z) \neq k)$, and show that this condition can co-exist with the conditions required for undelete as described in equation 35. In other words, we show that $undel(B) \wedge \{B^x \rightsquigarrow A\}_D \wedge \{\&A = \{B\}\}_D \wedge \{A^z\}_D \wedge (g(A^z) \neq k)$ can arise from valid file system execution.

We utilize the following implications for the proof:

$$\begin{aligned} \{B^x \rightsquigarrow A\}_D &\Leftrightarrow \{B^x \rightarrow A_k\}_M \prec \{\&A = \emptyset\}_M \prec write(B) \\ \{A^z\}_D &\Rightarrow \{c \rightarrow g(A^z)\}_M \prec write(A) \quad (\text{eq. 6}) \end{aligned}$$

Let us consider one possible interleaving of the above two event sequences:

$$\begin{aligned} \{c \rightarrow g(A^z)\}_M \prec write(A) \ll \{B^x \rightarrow A_k\}_M \prec \\ \{\&A = \emptyset\}_M \prec write(B) \end{aligned}$$

This is a valid file system sequence where a file represented by generation c points to $g(A^z)$, A^z is written to disk, then block A is freed from c thus killing the generation $g(A^z)$, and a new generation A_k of A is then allocated to the generation $g(B^x)$. Now, when $g(B^x)$ is deleted, and B is written to disk, the disk has both beliefs $\{B^x \rightsquigarrow A\}_D$ and $\{A^z\}_D$. Further, if the initial state of the disk was $\{\&A = \emptyset\}_D$, the above sequence would also simultaneously lead to the disk belief $\{\&A = \{B\}\}_D$. Thus we have shown that the conditions $\{B^x \rightsquigarrow A\}_D \wedge \{\&A = \{B\}\}_D \wedge \{A^z\}_D \wedge (k \neq g(A^z))$ can hold simultaneously. An undelete of B at this point would lead to violation of the consistency guarantee, because it would associate a stale generation of A with the undeleted file $g(B^x)$. It can be shown that neither reuse ordering nor pointer ordering would guarantee consistency in this case.

9.2 Undelete with generation pointers

We now propose the notion of *generation pointers* and show that with such pointers, consistent undelete is guaranteed. So far, we have assumed that *pointers* on disk point to *containers* (as discussed in Section 4). If instead, each pointer pointed to a specific *generation*, it leads to a different set of file system properties. To implement such “generation pointers”, each on-disk container contains a generation number that gets incremented every time the container is reused. In addition, every on-disk pointer will embed this generation number in addition to the container name. With generation pointers, the on-disk contents of a container will implicitly indicate its generation. Thus, $\{B_k\}_D$ is a valid belief; it means that the disk knows the contents of B belong to generation k .

Under generation pointers, the criterion for doing undelete (eq. 35) becomes:

$$\begin{aligned} undel(B) \wedge \{B^x \rightsquigarrow A_k\}_D \wedge \{A_k\}_D \\ \Leftrightarrow \{B^x \rightsquigarrow A_k\}_D \prec \{B^y \rightarrow A_k\}_D \end{aligned} \quad (36)$$

Let us introduce an additional constraint $\{A^z\}_D \wedge (k \neq g(A^z))$ into the left hand side of the above equation (as we did in the previous subsection):

$$\{B^x \rightsquigarrow A_k\}_D \wedge \{A_k\}_D \wedge \{A^z\}_D \wedge (k \neq g(A^z)) \quad (37)$$

Since $k \neq g(A^z)$, let us denote $g(A^z)$ as h . Since every on-disk container holds the generation number too, we have $\{A_h\}_D$. Thus, the above equation becomes:

$$\{B^x \rightsquigarrow A_k\}_D \wedge \{A_k\}_D \wedge \{A_h\}_D \wedge (k \neq h)$$

This is clearly a contradiction, since it means the on-disk container A has the two different generations k and h simultaneously. Thus, it follows that an undelete would not occur in this scenario (or alternatively, this would be flagged as inconsistent). Thus, all undeletes occurring under generation pointers are consistent.

9.3 Implementation of undelete in ext3

Following on the proof for consistent undelete, we implemented the generation pointer mechanism in Linux ext3. Each block has a generation number that gets incremented every time the block is reused. Although the generation numbers are maintained in a separate set of blocks, ensuring atomic commit of the generation number and the block data is straightforward in the data journaling mode of ext3, where we simply add the generation update to the create transaction. The block pointers in the inode are also extended with the generation number of the block. We implemented a tool for undelete that scans over the on-disk structures, restoring all files that can be undeleted *consistently*. Specifically, a file is restored if the generation information in all its metadata block pointers match with the corresponding block generation of the data blocks.

We ran a simple microbenchmark creating and deleting various directories from the linux kernel source tree, and observed that out of roughly 12,200 deleted files, 2970 files (roughly 25%) were detected to be inconsistent and not undeletable, while the remaining files were successfully undeleted. This illustrates that the scenario proved in Section 9.1 actually occurs in practice; an undelete tool without generation information would wrongly restore these files with corrupt or misleading data.

10 Application to Semantic Disks

An interesting application of a logic framework for file systems is that it enables reasoning about a recently proposed class of storage systems called “semantically-smart” disk systems (SDS). An SDS exploits file system information within the storage system, to provide better functionality [20]. However, as admitted by the authors [21], reasoning about the correctness of knowledge tracked in a semantic disk is quite hard. Our formalism of memory and disk beliefs fits the SDS model, since the extra file system state tracked by an SDS is essentially a disk belief. In this section, we first use our logic to explore the feasibility of tracking block type within a semantic disk.

We then show that the usage of generation pointers by the file system simplifies information tracking within an SDS.

10.1 Block typing

An important piece of information required within a semantic disk is the *type* of a disk container [21]. While identifying the type of statically-typed containers is straightforward, dynamically typed containers are hard to deal with. The type of a dynamically typed container is determined by the contents of a *parent* container; for example, an indirect pointer block can be identified only by observing a parent inode that has this block in its indirect pointer field. Thus, tracking dynamically typed containers requires correlating type information from a type-determining parent, and then using the information to interpret the contents of the dynamic container.

For accurate type detection in an SDS, we want the following guarantee to hold:

$$\{t(A^x) = k\}_D \Rightarrow \{t(A^x) = k\}_M \quad (38)$$

In other words, if the disk interprets the contents of an epoch A^x to be belonging to type k , those contents should have belonged to type k in memory as well. This guarantees, for example, that the disk would not wrongly interpret the contents of a normal data block container as an indirect block container. Note however that the equation does not impose any guarantee on *when* the disk identifies the type of a container; it only states that whenever it does, the association of type with the contents is correct.

To prove this, we first state an algorithm of how the disk arrives at a belief about a certain type [21]. An SDS snoops on metadata traffic, looking for type-determining containers such as inodes. When such a container is written, it observes the pointers within the container and concludes on the type of each of the pointers. Let us assume that one such pointer of type k points to container A . The disk then examines if container A was written since the last time it was freed. If yes, it interprets the current contents of A as belonging to type k . If not, when A is written at a later time, the contents are associated with type k . We have the following equation:

$$\begin{aligned} \{t(A^x) = k\}_D &\Rightarrow \{B^y \rightarrow A\}_D \wedge (f(B^y, A) = k) \\ &\wedge \{A^x\}_D \end{aligned} \quad (39)$$

In other words, to interpret A^x as belonging to type k , the disk must believe that some container B points to A , and the current on-disk epoch of B (*i.e.*, B^y) must indicate that A is of type k ; the function $f(B^y, A)$ abstracts this indication. Further, the disk must contain the contents of epoch A^x in order to associate the contents with type k .

Let us explore the logical events that should have led to each of the components on the right side of equation 39. Applying eq. 12,

$$\{B^y \rightarrow A\}_D \wedge (f(B^y, A) = k)$$

$$\begin{aligned} &\Rightarrow (\{B^y \rightarrow A\}_M \wedge (f(B^y, A) = k)) \prec \{B^y \rightarrow A\}_D \\ &\Rightarrow (\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D \end{aligned} \quad (40)$$

Similarly for the other component $\{A^x\}_D$,

$$\{A^x\}_D \Rightarrow \text{write}(A^x) \ll \{A^x\}_D \quad (41)$$

To verify the guarantee in equation 38, we assume that it does not hold, and then observe if it leads to a valid scenario. Thus, we can add the clause $\{t(A^x) = j\}_M \wedge (j \neq k)$ to equation 39, and our equation to prove is:

$$\{B^y \rightarrow A\}_D \wedge (f(B^y, A) = k) \wedge \{A^x\}_D \wedge \{t(A^x) = j\}_M$$

We thus have two event sequences (from eq. 40 and 41):

1. $(\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D$
2. $\{t(A^x) = j\}_M \wedge \text{write}(A^x)$

Since the type of an epoch is unique, and a *write* of a container implies that it already has a type,

$$\{t(A^x) = j\}_M \wedge \text{write}(A^x) \Rightarrow \{t(A^x) = j\}_M \prec \text{write}(A^x).$$

These sequences can only be interleaved in two ways. The epoch A^x occurs either before or after the epoch in which $\{t(A) = k\}_M$.

Interleaving 1:

$$\begin{aligned} &(\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D \\ &\ll \{t(A^x) = j\}_M \prec \text{write}(A^x) \end{aligned}$$

By eq. 11,

$$\begin{aligned} &\Rightarrow (\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D \\ &\ll \{\&A = \emptyset\}_M \prec \{t(A^x) = j\}_M \prec \text{write}(A^x) \end{aligned}$$

This is a valid sequence where the container A is freed after the disk acquired the belief $\{B \rightarrow A\}$ and a later version of A is then written when its actual type has changed to j in memory, thus leading to incorrect interpretation of A^x as belonging to type k .

However, in order to prevent this scenario, we simply need the reuse ordering rule (eq. 14). With that rule, the above sequence would imply the following:

$$\begin{aligned} &\Rightarrow (\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D \\ &\ll \{\&A = \emptyset\}_M \prec \text{write}(B) \ll \{t(A^x) = j\}_M \prec \text{write}(A^x) (\{B^y \rightarrow A_g\}_M \wedge \{t(A_g) = k\}_M) \prec \{B^y \rightarrow A_g\}_D \\ &\Rightarrow (\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D \\ &\ll \{\&A = \emptyset\}_D \prec \{t(A^x) = j\}_M \prec \text{write}(A^x) \end{aligned}$$

Thus, when A^x is written, the disk will be treating A as free, and hence will not wrongly associate A with type k .

Interleaving 2:

Proceeding similarly with the second interleaving where epoch A^x occurs before A is assigned type k , we arrive at the following sequence:

$$\begin{aligned} &\Rightarrow \{t(A^x) = j\}_M \prec \text{write}(A^x) \ll \{\&A = \emptyset\}_M \\ &\prec (\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D \end{aligned}$$

We can see that simply applying the reuse ordering rule does not prevent this sequence. We need a stronger form

of reuse ordering where the “freed state” of A includes not only the containers that pointed to A , but also the allocation structure $|A|$ tracking liveness of A . With this rule, the above sequence becomes:

$$\begin{aligned} &\Rightarrow \{t(A^x) = j\}_M \prec \text{write}(A^x) \ll \{\&A = \emptyset\}_M \\ &\prec \text{write}(|A|) \ll (\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \\ &\prec \{B^y \rightarrow A\}_D \end{aligned} \quad (42)$$

We also need to add a new behavior to the SDS which states that when the SDS observes an allocation structure indicating that A is free, it inherits the belief that A is free.

$$\{\&A = \emptyset\}_M \prec \text{write}(|A|) \Rightarrow \{\&A = \emptyset\}_D$$

Applying the above SDS operation to eqn 42, we get

$$\begin{aligned} &\Rightarrow \{t(A^x) = j\}_M \prec \text{write}(A^x) \ll \{\&A = \emptyset\}_D \\ &\ll (\{B^y \rightarrow A\}_M \wedge \{t(A) = k\}_M) \prec \{B^y \rightarrow A\}_D \end{aligned}$$

In this sequence, because the SDS does not observe a write of A since it was treated as “free”, it will not associate type k to A until A is subsequently written.

Thus, we have shown that an SDS cannot accurately track dynamic type underneath a file system without any ordering guarantees. We have also shown that if the file system exhibits a strong form of reuse ordering, dynamic type detection can indeed be made reliable within an SDS.

10.2 Utility of generation pointers

In this subsection, we explore the utility of file system-level “generation pointers” (§ 9.2) in the context of an SDS. To illustrate their utility, we show that tracking dynamic type in an SDS is straightforward if the file system tracks generation pointers.

With generation pointers, equation 39 becomes:

$$\begin{aligned} \{t(A_g) = k\}_D &\Rightarrow \{B^y \rightarrow A_g\}_D \wedge (f(B^y, A_g) = k) \\ &\wedge \{A_g\}_D \end{aligned}$$

The two causal event sequences (as explored in the previous subsection) become:

$$\{t(A_g) = j\}_M \wedge \text{write}(A_g)$$

Since the above sequences imply that the same generation g had two different types, it violates rule 11. Thus, we straightaway arrive at a contradiction that proves that violation of rule 38 can never occur.

11 Related Work

Previous work has recognized the need for modeling complex systems with formal frameworks, in order to facilitate proving correctness properties about them. The logical framework for reasoning about authentication protocols, proposed by Burrows *et al.* [4], is the most related

to our work in spirit; in that paper, the authors formulate a domain-specific logic and proof system for authentication, showing that protocols can be verified through simple logical derivations. Other domain-specific formal models exist in the areas of database recovery [9] and database reliability [7].

A different body of related work involves generic frameworks for modeling computer systems. The well-known TLA framework [10] is an example. The I/O automaton [1] is another such framework. While these frameworks are general enough to model most complex systems, their generality is also a curse; modeling various aspects of a file system to the extent we have in this paper, is quite tedious with a generic framework. Tailoring the framework by using domain-specific knowledge makes it simpler to reason about properties using the framework, thus significantly lowering the barrier to entry in terms of adopting the framework [4]. Specifications and proofs in our logic take 10 to 20 lines in contrast to the thousands of lines that TLA specifications take [25]. However, automated theorem-proving through model checkers is one of the benefits of using a generic framework such as TLA.

Previous work has also explored verification of the correctness of system *implementations*. The recent body of work on using model checking to verify implementations is one example [14, 24]. We believe that this body of work is complementary to our logic framework; our logic framework can be used to build the model and the invariants that should hold in the model, which the implementation can be verified against.

Finally, the file system properties we have listed in Section 6 have been identified in previous work on soft updates [6] and more recent work on semantic disks [20].

12 Conclusions

As the need for dependability of computer systems becomes more important than ever, it is essential to have systematic formal frameworks to verify and reason about their correctness. Despite file systems being a critical component of system dependability, formal verification of their correctness has been largely ignored. Besides making file systems vulnerable to hidden errors, the absence of a formal framework also stifles innovation, because of the skepticism towards the correctness of new proposals, and the proclivity to stick to “time-tested” alternatives. In this paper, we have taken a step towards bridging this gap in file system design by showing that a logical framework can substantially simplify and systematize the process of verifying file system correctness.

Acknowledgements

We would like to thank Lakshmi Bairavasundaram, Nathan Burnett, Timothy Denehy, Rajasekar Krishnamurthy, Florentina Popovici, Vijayan Prabhakaran, and Vinod Yegneswaran for their comments on earlier drafts

of this paper. We also thank the anonymous reviewers for their excellent feedback and comments, many of which have greatly improved this paper.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0325267, IBM, Network Appliance, and EMC.

References

- [1] P. C. Attie and N. A. Lynch. Dynamic Input/Output Automata, a Formal Model for Dynamic Systems. In *ACM PODC*, 2001.
- [2] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.
- [3] N. Björner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. E. Uribe. Verifying temporal properties of reactive systems: A STeP tutorial. *Formal Methods in System Design (FMSD)*, 16(3):227–270, 2000.
- [4] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. In *ACM SOSP*, pages 1–13, 1989.
- [5] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.
- [6] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM TOCS*, 18(2), May 2000.
- [7] V. Hadzilacos. A Theory of Reliability in Database Systems. *J. ACM*, 35(1):121–145, 1988.
- [8] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, Nov. 1987.
- [9] D. Kuo. Model and Verification of a Data Manager Based on ARIES. *ACM Trans. Database Systems*, 21(4):427–479, 1996.
- [10] L. Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
- [11] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.
- [12] J. C. Mogul. A Better Update Policy. In *USENIX Summer '94*, Boston, MA, June 1994.
- [13] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. S. z. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, March 1992.
- [14] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI '02*, Dec. 2002.
- [15] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science (TCS)*, 13:45–60, 1981.
- [16] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *USENIX '05*, 2005.
- [17] R-Undelete. R-Undelete File Recovery Software. <http://www.r-undelete.com/>.
- [18] H. Reiser. ReiserFS. www.namesys.com, 2004.
- [19] Restorer2000. Restorer 2000 Data Recovery Software. <http://www.bitmart.net/>.
- [20] M. Sivathanu, L. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *OSDI '04*, pages 379–394, San Francisco, CA, December 2004.
- [21] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRaid. In *FAST04*, 2004.
- [22] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.
- [23] S. C. Tweedie. EXT3, Journaling File System. <http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html>, July 2000.
- [24] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. In *OSDI '04*, Dec. 2004.
- [25] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. *Lecture Notes in Computer Science*, (1703):54–66, 1999.

Providing Tunable Consistency for a Parallel File Store

Murali Vilayannur¹ Partho Nath Anand Sivasubramaniam
Department of Computer Science and Engineering
Pennsylvania State University
{vilayann,nath,anand}@cse.psu.edu

Abstract

Consistency, throughput, and scalability form the backbone of a cluster-based parallel file system. With little or no information about the workloads to be supported, a file system designer has to often make a one-glove-fits-all decision regarding the consistency policies. Taking a hard stance on consistency demotes throughput and scalability to second-class status, having to make do with whatever leeway is available. Leaving the choice and granularity of consistency policies to the user at open/mount time provides an attractive way of providing the best of all worlds. We present the design and implementation of such a file-store, CAPFS (Content Addressable Parallel File System), that allows the user to define consistency semantic policies at runtime. A client-side plug-in architecture based on user-written plug-ins leaves the choice of consistency policies to the end-user. The parallelism exploited by use of multiple data stores provides for bandwidth and scalability. We provide extensive evaluations of our prototype file system on a concurrent read/write workload and a parallel tiled visualization code.

1 Introduction

High-bandwidth I/O continues to play a critical role in the performance of numerous scientific applications that manipulate large data sets. Parallelism in disks and servers provides cost-effective solutions at the hardware level for enhancing I/O bandwidth. However, several components in the system software stack, particularly in the file system layer, fail to meet the demands of applications. This is primarily due to tradeoffs that parallel file system designers need to make between performance and scalability goals at one end, and transparency and ease-of-use goals at the other.

Compared to network file systems (such as NFS [26], AFS [14], and Coda [16]), which despite allowing multiple file servers still allocate all portions of a file to a server, parallel file systems (such as PVFS [7],

GPFS [27], and Lustre [6]) distribute portions of a file across different servers. With the files typically being quite large and different processes of the same application sharing a file, such striping can amplify the overall bandwidth. With multiple clients reading and writing a file, coordination between the activities becomes essential to enforce a consistent view of the file system state.

The level of sharing when viewed at a file granularity in parallel computing environments is much higher than that observed in network file systems [4, 21], making consistency more important. Enforcement of such consistency can, however, conflict with performance and scalability goals. Contemporary parallel file system design lacks a consensus on which path to take. For instance, PVFS provides high-bandwidth access to I/O servers without enforcing overlapping-write atomicity, leaving it entirely to the applications or runtime libraries (such as MPI-I/O [9]) to handle such consistency requirements. On the other hand, GPFS and Lustre enforce byte-range POSIX [28] consistency. Locking is used to enforce serialization, which in turn may reduce performance and scalability (more scalable strategies are used in GPFS for fine-grained sharing, but the architecture is fundamentally based on distributed locking).

Serialization is not an evil but a necessity for certain applications. Instead of avoiding consistency issues and using an external mechanism (e.g., DLM [15]) to deal with serialization when required, incorporating consistency enforcement in the design might reduce the overheads. Hence the skill lies in being able to make an informed decision regarding the consistency needs of an application. A key insight here is that applications, not the system, know best to deal with their concurrency needs. In fact, partial attempts at such optimizations already exist — many parallel applications partition the data space to minimize read-write and write-write sharing. Since different applications can have different sharing behavior, designing for performance *and* consistency would force the design to cater to *all* their needs — simultaneously! Provisioning a single (and strict) consistency mechanism may not only make such fine-grained

customization hard but may also constrain the suitability of running diverse sets of applications on the same parallel file system.

Addressing some of these deficiencies, this paper presents the design and implementation of a novel parallel file system called CAPFS that provides the following notable features:

- To the best of our knowledge, CAPFS is the first file system to provide a tunable consistency framework that can be customized for an application. A set of plug-in libraries is provided with clearly defined entry points, to implement different consistency models, including POSIX, Session, and Immutable-files. Though a user could build a model for each application, we envision a set of predefined libraries that an application can pick before execution for each file and/or file system.
- The data store in CAPFS is content-addressable. Consequently, blocks are not modified in place, allowing more concurrency in certain situations. In addition, content addressability can make *write propagation* (which is needed to enforce coherence) more efficient. For instance, update-based coherence mechanisms are usually avoided because of the large volume of data that needs to be sent. In our system however, we allow update messages that are just a sequence of (cryptographic) hashes of the new content being generated. Further, content addressability can exploit commonality of content within and across files, thereby lowering caching and network bandwidth requirements.
- Rather than locking when enforcing serialization for read-write sharing or write-write sharing (write atomicity), CAPFS uses optimistic concurrency control mechanisms [17, 19] with the presumption that these are rare events. Avoidance of distributed locking enhances the scalability and fault-tolerance of the system.

The rest of this paper is organized as follows. The next section outlines the design issues guiding our system architecture, following which the system architecture and the operational details of our system are presented in Section 3. An experimental evaluation of our system is presented in Section 4 on a concurrent read/write workload and on a parallel tiled visualization code. Section 5 summarizes related work and Section 6 concludes with the contributions of this paper and discusses directions for further improvements.

2 Design Issues

The guiding rails of the CAPFS design is based on the following goals: 1) user should be able to define the consistency policy at a chosen granularity, and 2) implementation of consistency policies should be as lightweight and concurrent as possible. The CAPFS design explores these directions simultaneously — providing easily expressible, tunable, robust, lightweight and scalable con-

sistency without losing focus of the primary goal of providing high bandwidth.

2.1 Tunable Consistency

If performance is a criterion, consistency requirements for applications might be best decided by applications themselves. Forcing an application that has little or no sharing to use a strong or strict consistency model may lead to unnecessarily reduced I/O performance. Traditional techniques to provide strong file system consistency guarantees for both meta-data and data use variants of locking techniques. In this paper, we are interested in providing tunable semantic guarantees for *file data alone*.

The choice of a system wide consistency policy may not be easy. NFS [26] offers poorly defined consistency guarantees that are not suitable for parallel workloads. On the other hand, Sprite [20] requires the central server to keep track of all concurrent sessions and disable caching at clients when write-sharing is detected. Such an approach forces *all* write-traffic to be network bound from thereon until one or more processes close the shared file. Although such a policy enforces correctness, it penalizes performance of applications when writers update spatially disjoint portions of the same file which is quite common in parallel workloads. For example, an application may choose to have a few temporary files (store locally, no consistency), a few files that it knows no one else will be using (no consistency), a few files that will be extensively shared (strong consistency), and a few files that might have sharing in the rare case (weaker user-defined consistency). A single consistency policy for a cluster-based file system cannot cater to the performance of different workloads such as those described above.

As shown in Figure 1, CAPFS provides a client-side plug-in architecture to enable users to define their own consistency policies. The users write plug-ins that define what actions should be taken before and after the client-side daemon services the corresponding system call. (The details of the above mechanism are deferred to Section 3.6).

The choice of a plug-in architecture to implement this functionality has several benefits. Using this architecture, a user can define not just standard consistency policies like POSIX, session and NFS, but also custom policies, at a chosen granularity (sub-file, file, partition-wide). First and foremost, the client keeps track of its files; servers do not need to manage copy-sets unless explicitly requested by client. Furthermore, a client can be using several different consistency policies for different files or even changing the consistency policy for a given file *at runtime*, without having to recompile or restart the file system or even the client-side daemon (Figure 1). All that is needed is that a desired policy be compiled as a plug-in and be installed in a special directory, after which the daemon is sent a signal to indicate the availability of

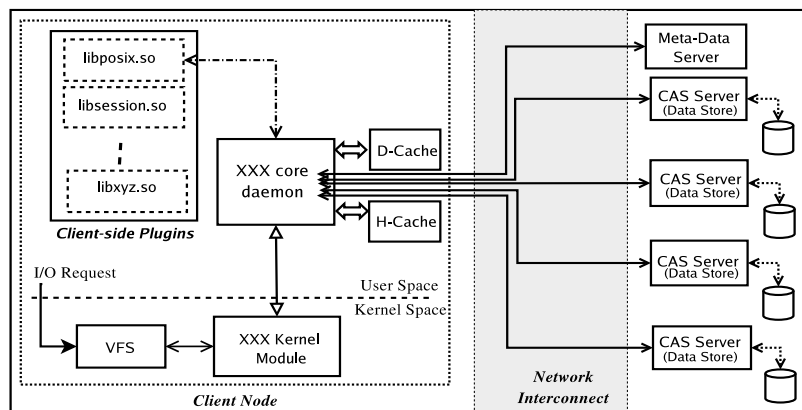


Figure 1: Design of the CAPFS parallel file system: Clients feature a user-space plug-in architecture, a content-addressable data cache(D-Cache), and an H-Cache(hash cache) to cache hashes of data blocks encountered.

a new policy. Leaving the choice of the consistency policy *and* allowing the user to change it at runtime enable tuning performance at a very fine granularity. However, one major underlying assumption in our system design is that we anticipate that the file system administrator sets the same policy on all the nodes of the cluster that accesses the file system. Handling conflicting consistency policies for the same file system or files could lead to incorrect execution of applications.

2.2 Lightweight Synchronization

Any distributed file system needs to provide a consistency protocol to arbitrate accesses to data and meta-data blocks. The consistency protocol needs to expose primitives both for atomic read/modify/write operations and for notification of updates to regions that are being managed. The former primitive is necessary to ensure that the state of the system is consistent in the presence of multiple updates, while the latter is necessary to incorporate client caching and prevent stale data from being read. Traditional approaches use locking to address both these issues.

2.2.1 To Lock or Not to Lock?

Some parallel cluster file systems (such as Lustre [6] and GPFS [27]) enforce data consistency by using file locks to prevent simultaneous file access from multiple clients. In a networked file system, this strategy usually involves acquiring a lock from a central lock manager on a file before proceeding with the write/read operation. Such a coarse-grained file locks-based approach ensures that only one process at a time can write data to a file. As the number of processes writing to the same file increases, performance (from lock contention) degrades rapidly. On the other hand, fine-grained file-locking schemes, such as byte-range locking, allow multiple processes to simultaneously write to different regions of a shared file. However, they also restrict scalability because of the overhead associated with maintaining state for a large number of locks, eventually lead-

ing to performance degradation. Furthermore, any networked locking system introduces a bottleneck for data access: the lock server.

The recent explosion in the scale of clusters, coupled with the emphasis on fault tolerance, has made traditional locking less suitable. GPFS [27], for instance, uses a variant of a distributed lock manager algorithm that essentially runs at two levels: one at a central server and the other on every client node. For efficiency reasons, clients can cache lock tokens on their files until they are explicitly revoked.

Such optimizations usually have hidden costs. For example, in order to handle situations where clients terminate while holding locks, complex lock recovery/release mechanisms are used. Typically, these involve some combination of a distributed crash recovery algorithm or a lease system [11]. Timeouts guarantee that lost locks can be reclaimed within a bounded time. Any lease-based system that wishes to guarantee a sequentially consistent execution must handle a race condition, where clients must finish their operation after acquiring the lock before the lease terminates. Additionally, the choice of the lease timeout is a tradeoff between performance and reliability concerns and further exacerbates the problem of reliably implementing such a system.

The pitfalls of using locks to solve the consistency problems in parallel file systems motivated us to investigate different approaches to providing the same functionality. We use a lockless approach for providing atomic file system data accesses. The approach to providing lockless, sequentially consistent data in the presence of concurrent conflicting accesses presented here has roots in three other transactional systems: store conditional operations in modern microprocessors [18], optimistic concurrency algorithms in databases [17], and optimistic concurrency approach in the Amoeba distributed file service [19].

Herlihy [13] proposed a methodology for constructing lock-free and wait-free implementations for highly concurrent objects using the load-linked and store-conditional instructions. Our lockless approach, similar

in spirit, does not imply the absence of any synchronization primitives (such as barriers) but, rather, implies the *absence of a distributed byte-range file locking service*. By taking an optimistic approach to consistency, we hope to gain on concurrency and scalability, while pinning our bets on the fact that conflicting updates (write-sharing) will be rare [4, 8, 21]. In general, it is well understood that optimistic concurrency control works best when updates are small or when the probability of simultaneous updates to the same item is small [19]. Consequently, we expect our approach to be ideal for parallel scientific applications. Parallel applications are likely to have each process write to distinct regions in a single shared file. For these types of applications, there is no need for locking, and we would like for all writes to proceed in parallel without the delay introduced by such an approach.

2.2.2 Invalidates or Updates?

Given that client-side caching is a proven technique with apparent benefits for a distributed file system, a natural question that arises in the context of parallel file systems is whether the cost of keeping the caches coherent outweighs the benefits of caching. However, as outlined earlier, we believe that deciding to use caches and whether to keep them coherent should be the prerogative of the consistency policy and should not be imposed by the system. Thus, only those applications that require strict policies and cache coherence are penalized, instead of the whole file system. A natural consequence of opting to cache is the mechanism used to synchronize stale caches; that is, should consistency mechanisms for keeping caches coherent be based on expensive update-based protocols or on cheaper invalidation-based protocols or hybrid protocols?

Although update-based protocols reduce lookup latencies, they are not considered a suitable choice for workloads that exhibit a high degree of read-write sharing [3]. Furthermore, an update-based protocol is inefficient in its use of network bandwidth for keeping file system caches coherent, thus leading to a common adoption of invalidation-based protocols.

As stated before, parallel workloads do not exhibit much block-level sharing [8]. Even when sharing does occur, the number of consumers that actually read the modified data blocks is typically low. In Figure 2 we compute the number of consumers that read a block between two successive writes to the same block (we assume a block size of 4 KB). Upon normalizing against the number of times sharing occurs, we get the values plotted in Figure 2. This figure was computed from the traces of four parallel applications that were obtained from [31]. In other words, Figure 2 attempts to convey the amount of read-write sharing exhibited by typical parallel applications. It indicates that the number of consumers of a newly written block is very small (with the exception of LU, where a newly written block is read by

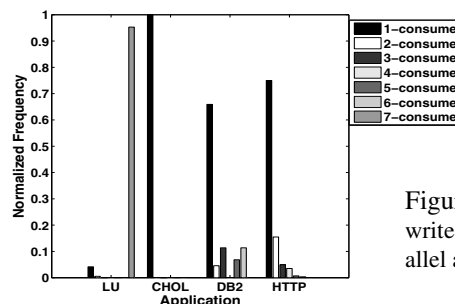


Figure 2: Read-write sharing for parallel applications.

all the remaining processes before the next write to the same block). Thus, an update-based protocol may be viable as long as the update mechanism does not consume too much network bandwidth. This result motivated us to consider content-addressable cryptographic hashes (such as SHA-1 [12]) for maintaining consistency because they allow for a bandwidth-efficient update-based protocol by transferring just the hash in place of the actual data. We defer the description of the actual mechanism to Section 3.5.

2.2.3 Content Addressability

Content addressability provides an elegant way to summarize the contents of a file. It provides the following advantages:

- The contents of a file can be listed as a concatenation of the hashes of its blocks. Such a representation was referred to as *recipes* in a previous study [30]. This approach provides a lightweight method of updating or invalidating sections of a file and so forth.
- It increases system concurrency, by not requiring synchronization at the content-addressable data servers (Figure 1). In comparison to versioning file systems that require a central version/time-stamp servers [19] or a distributed protocol for obtaining unique timestamps [10], a content-addressable system provides an independent, autonomous technique for clients to generate new version numbers for a block. Since newly written blocks will have new cryptographic checksums (assuming no hash collisions), a content-addressable data server also achieves the “no-overwrite” property that is essential for guaranteeing any sort of consistency.
- Using cryptographic hashes also allows for a bandwidth-efficient update-based protocol for maintaining cache coherence. This forms the basis for adopting a content-addressable storage server design in place of a traditional versioning mechanism. Additionally, it is foreseeable that the content-addressable nature of data may lead to easy replication schemes.
- Depending on the workload, content addressability might be able to reduce network traffic and storage demands. Blocks with the same content, if in the cache (because of commonality of data across files or within

a file) do not need to be fetched or written. Only a single instance of the common block needs to be stored, leading to space savings.

As shown in Figure 1, the client employs two caches for performance. The H-Cache, or hash cache, stores all or a portion of a file's *recipe* [30]. A file in the CAPFS file system is composed of content-addressable chunks. Thus, a chunk is the unit of computation of cryptographic hashes and is also the smallest unit of accessibility from the CAS servers. The chunk size is crucial because it can impact the performance of the applications. Choosing a very small value of chunk size increases the CPU computation costs on the clients and the overheads associated with maintaining a large recipe file, while a very large value of chunk size may increase the chances of false sharing and hence coherence traffic. Thus, we leave this as a tunable knob that can be set by the plug-ins at the time of creation of a file and is a part of the file's meta-data. For our experiments, unless otherwise mentioned, we chose a default chunk size of 16 KB. The recipe holds the mapping between the chunk number and the hash value of the chunk holding that data. Using the H-Cache provides a lightweight method of providing updates when sharing occurs. An update to the hashes of a file ensures that the next request for that chunk will fetch the new content.

The D-Cache, or the data cache, is a content addressable cache. The basic object stored in the D-Cache is a chunk of data addressed by its SHA1-hash value. One can think of a D-cache as being a local replica of the CAS server's data store. When a section of a file is requested by the client, the corresponding data chunks are brought into the D-Cache. Alternatively, when the client creates new content, it is also cached locally in the D-Cache. The D-Cache serves as a simple cache with *no consistency requirements*. Since the H-caches are kept coherent (whenever the policy dictates), there is no need to keep the D-caches coherent. Additionally, given a suitable workload, it could also exploit commonality across data chunks and possibly across temporal runs of the same benchmark/application, thus potentially reducing latency and network traffic.

3 System Architecture

The goal of our system is to provide a robust parallel file system with good concurrency, high throughput and tunable consistency. The design of CAPFS resembles that of PVFS [7] in many aspects — central meta-data server, multiple data servers, RAID-0-style striping of data across the I/O servers, and so forth. The RAID-0 striping scheme also enables a client to easily calculate which data server has which data blocks of a file. In this section, we first take a quick look at the PVFS architecture and its limitations from the perspective of consistency semantics and then detail our system's design.

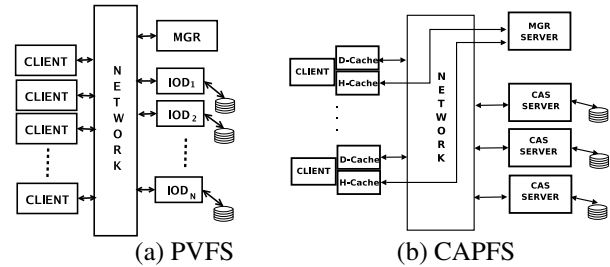


Figure 3: System architectures: CAPFS design incorporates two client-side caches that are absent in PVFS.

Figure 3 depicts a simplified diagram of the PVFS and CAPFS system architectures.

3.1 PVFS Architecture

The primary goal of PVFS as a parallel file system is to provide high-speed access to file data for parallel applications. PVFS is designed as a client-server system, as shown in Figure 3 (a).

PVFS uses two server components, both of which run as user-level daemons on one or more nodes of the cluster. One of these is a meta-data server (called MGR) to which requests for meta-data management (access rights, directories, file attributes, and physical distribution of file data) are sent. In addition, there are several instances of a data server daemon (called IOD), one on each node of the cluster whose disk is being used to store data as part of the PVFS name space. There are well-defined protocol structures for exchanging information between the clients and the servers. For instance, when a client wishes to open a file, it communicates with the MGR daemon, which provides it the necessary meta-data information (such as the location of IOD servers for this file, or stripe information) to do subsequent operations on the file. Subsequent reads and writes to this file do not interact with the MGR daemon and are handled directly by the IOD servers.

This strategy is key to achieving scalable performance under concurrent read and write requests from many clients and has been adopted by more recent parallel file system efforts. However, a flip-side to this strategy is that the file system does not guarantee any data consistency semantics in the face of conflicting operations or sessions. Fundamental problems that need to be addressed to offer sequential/ POSIX [28] style semantics are the *write atomicity* and *write propagation* requirements. Since file data is striped across different nodes and since the data is always overwritten, the I/O servers cannot guarantee write atomicity, and hence reads issued by clients could contain mixed data that is disallowed by POSIX semantics. Therefore, any application that requires sequential semantics must rely on external tools or higher-level locking solutions to enforce access restrictions. For instance, any application that relies on UNIX/POSIX semantics needs to use a distributed

cluster-wide lock manager such as the DLM [15] infrastructure, so that all `read/write` accesses acquire the appropriate file/byte-range locks before proceeding.

3.2 CAPFS: Servers

The underlying foundation for our system is the content-addressable storage model, wherein file blocks are *addressed and located* based on the cryptographic hashes of their contents. A file is logically split into fixed-size data chunks, and the hashes for these chunks are stored in the *hash server daemon*. The hash server daemon, analogous to the meta-data server (MGR) daemon of the PVFS system design, is responsible for mapping and storing the hashes of file blocks (termed recipes [30]) for all files. In essence, this daemon translates the logical block-based addressing mode to the content addressable scheme, that is, given a logical block i of a particular file F , the daemon returns the hashes for that particular block. Even though in the current implementation there is a central server, work is under way to use multiple meta-data servers to serve a file's hashes for load-balancing purposes. Throughout the rest of the paper, we will use the term MGR server synonymously with hash server or meta-data server to refer to this daemon.

Analogous to the PVFS I/O server daemon is a content-addressable server (CAS) daemon, which supports a simple *get/put* interface to retrieve/store data blocks based on their cryptographic hashes. However, this differs significantly both in terms of functionality and exposed interfaces from the I/O servers of PVFS. Throughout the rest of this paper, we will use the term CAS server synonymously with data server to refer to this daemon.

3.3 CAPFS: Clients

The design of the VFS glue in CAPFS is akin to the upcall/downcall mechanism that was initially prototyped in the Coda [16] file system (and later adapted in many other file systems including PVFS). In this design, file system requests obtained from the VFS are queued in a device file and serviced by a user-level daemon. If an error is generated or if the operation completes successfully, the response is queued back into the device file, and the kernel signals the process that was waiting for completion of the operation. The client-side code intercepts these upcalls and funnels meta-data operations to the meta-data server. The data operations are striped to the appropriate CAS servers. Prototype implementations of the VFS glue are available at [1] for both Linux 2.4 and 2.6 kernels.

3.4 System Calls

The CAPFS system uses optimistic concurrency mechanisms to handle write atomicity on a central meta-

data server, while striping writes in parallel over multiple content-addressable servers (CAS servers). The system has a lockless design: the only form of locking used is mutual-exclusion locks on the meta-data server to serialize the multiple threads (whenever necessary), as opposed to distributed locking schemes (such as DLM [15]).

3.4.1 Steps for the `open` and `close` System Call

- When a client wishes to open a file, a request is sent to the hash-server to query the hashes for the file if any.
- The server returns the list of hashes for the file (if the file is small). Hashes can also be obtained on demand from the server subsequently. The server also adds H-cache callbacks to this node for this file if requested.
- After the hashes are obtained, the client caches them locally (if specified by the policy) in the H-cache to minimize server load. H-cache coherence is achieved by having the server keep track of when commits are successful, and issuing callbacks to clients that may have cached the hashes. This step is described in greater detail in the subsequent discussions.
- On the last close of the file, all the entries in the H-cache for this file are invalidated for subsequent opens to reacquire, and if necessary the server is notified to terminate any callbacks for this node.

3.4.2 Steps for the `read` System Call

- The client tries to obtain the appropriate hashes for the relevant blocks either from the H-cache or from the hash server. An implicit agreement here is that the server promises to keep the client's H-cache coherent. This goal may be achieved by using either an update-based mechanism or an invalidation-based mechanism depending on the number of sharers. Note that the update callbacks contain merely the hashes and not the actual data.
- Using these hashes, it tries to locate the blocks in the D-cache. Note that keeping the H-cache coherent is enough to guarantee sequential consistency; nothing needs to be done for the D-cache because it is content addressable.
- If the D-cache has the requested blocks, the read returns and the process continues. On a miss, the client issues a *get* request to the appropriate CAS servers, which is cached subsequently. Consequently, reads in our system do not suffer any slowdowns and should be able to exploit the available bandwidth to the CAS servers by accessing data in parallel.

3.4.3 Steps for the `write` System Call

Writes from clients need to be handled a little differently because consistency guarantees may have to be met (depending on the policy). Since writes change the contents of the block, the cryptographic hashes for the block changes, and hence this is a new block in the system

altogether. We emphasize that we need mechanisms to ensure write atomicity not only across blocks but also across copies that may be cached on the different nodes. On a write to a block, the client does the following sequence of steps,

- Hashes for all the relevant blocks are obtained either from the H-cache or from the hash server.
- If the write spans an entire block, then the new hash can be computed locally by the client. Otherwise, it must read the block and compute the new hash based on the block's locally modified contents.
- After the old and new hashes for all relevant blocks are fetched or computed, the client does an *optimistic put* of the new blocks to the CAS servers, which store the new blocks. Note that by virtue of using content-addressable storage, the servers do not overwrite older blocks. This is an example of an optimistic update, because we assume that the majority of writes will be race-free and uncontested.
- If the policy requires that the writer's updates be made immediately visible, the next step is the *commit* operation. Depending on the policy, the client informs the server whether the commit should be forced or whether it can fail. Upon a successful commit, the return values are propagated back.
- A failed commit raises the possibility of *orphaned* blocks that have been stored in the I/O servers but are not part of any file. Consequently, we need a distributed cleaner process that is invoked when necessary to remove blocks that do not belong to any file. We refer readers to [1] for a detailed description of the cleaner protocol.

3.4.4 Commit Step

- In the commit step, the client contacts the hash server with the list of blocks that have been updated, the set of old hashes, and the set of new hashes. In the next section, we illustrate the need for sending the old hashes, but in short they are used for detecting concurrent write-sharing scenarios similar to store-conditional operations [18].
- The meta-data server atomically compares the set of old hashes that it maintains with the set of old hashes provided by the client. In the uncontested case, all these hashes would match, and hence the commit is deemed race free and successful. The hash server can now update its recipe list with the new hashes. In the rare case of a concurrent conflicting updates, the server detects a mismatch in the old hashes reported for one or more of the client's commits and asks them to retry the entire operation. However, clients can override this by requesting the server to force the commit despite conflicts.
- Although such a mechanism has guaranteed write-atomicity across blocks, we still need to provide mechanisms to ensure that client's caches are also updated or invalidated to guarantee write atomicity across all copies of blocks that may be required by the consistency

policy (sequential consistency/UNIX semantics require this). Since the server keeps track of clients that may have cached file hashes, a successful commit also entails updating or invalidating any client's H-cache with the latest hashes.

- Our system guarantees that updates to all locations are made visible in the same order to all clients (this mechanism is not exposed to the policies yet). Therefore, care must be exercised in the previous step to ensure that updates to all clients' H-caches are atomic. In other words, if multiple clients may have cached the hashes for a particular chunk and if the hash-server decides to update the hashes for the same chunk, the update-based protocol must use a two-phase commit protocol (such as those used in relational databases), so that all clients see the updates in the same order. This is not needed in an invalidation-based protocol however. Hence, we use an invalidation-based protocol in the cases of multiple readers/writers and an update-based protocol for single reader/writer scenarios.

3.5 Conflict Resolution

Figure 4 depicts a possible sequence of actions and messages that are exchanged in the case of multiple-readers and a single-writer client to the same file. We do not show the steps involved in opening the file and caching the hashes. In step 1, the writer optimistically writes to the CAS servers after computing the hashes locally. Step 2 is the request for committing the write sent to the hash server. Step 3 is an example of the invalidation-based protocol that is used in the multiple reader scenario from the point of view of correctness as well as performance. Our system resorts to an update-based protocol in the single sharer case. Sequential consistency requires that any update-based protocol has to be two-phased for ensuring the write-ordering requirements, and hence we opted to dynamically switch to using invalidation-based protocol in this scenario to alleviate performance concerns. Steps 5 and 6 depict the case where the readers look up the hashes and the local cache. Since the hashes could be invalidated by the writer, this step may also incur an additional network transaction to fetch the latest hashes for the appropriate blocks. After the hashes are fetched, the reader looks up its local data cache or sends requests to the appropriate data servers to fetch the data blocks. Steps 5 and 6 are shown in dotted lines to indicate the possibility that a network transaction may not be necessary if the requested hash and data are cached locally (which happens if both the *read*'s occurred before the *write* in the total ordering).

Figure 5 depicts a possible sequence of actions and messages that are exchanged in the case of multiple-readers and multiple-writers to the same file. As before, we do not show the steps involved in opening the file and caching the hashes. In step 1, writer client II optimistically writes to the CAS servers after computing

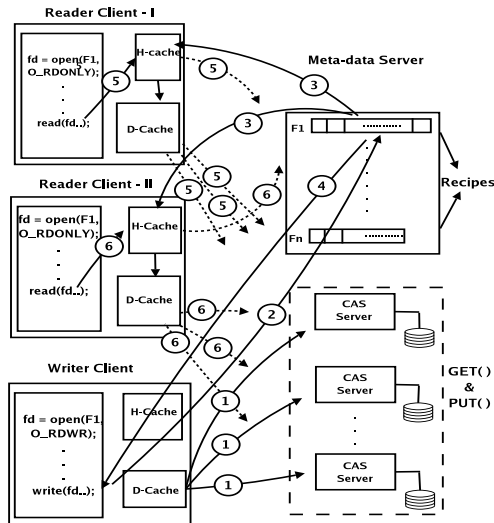


Figure 4: Action sequence: multiple-readers single-writer

hashes locally. In step 2, writer client I does the same after computing hashes locally. Both these writers have at least one overlapping byte in the file to which they are writing (*true-sharing*) or are updating different portions of the same chunk (*false-sharing*). In other words this is an instance of concurrent-write sharing. Since neither writer is aware of the other's updates, one of them is asked to retry. The hash server acts as a serializing agent. Since it processes requests from client II before client I, the write from client II is successfully committed, and step 3 shows the invalidation messages sent to the reader and the writer client. Step 4 is the acknowledgment for the successful write commit. Step 5 is shown dashed to indicate that the hash server requests writer client I to retry its operation. The write done by this client in step 2 is shown dotted to indicate that this created orphaned blocks on the data server and needs to be cleaned. After receiving a reply from the hash server that the write needs to be retried, the writer client I obtains the latest hashes or data blocks to recompute its hashes and reissues the write as shown in step 6.

In summary, our system provides mechanisms to achieve serializability that can be used by the consistency policies if they desire. In our system, *read-write serializability* and *write atomicity across copies* are achieved by having the server update or invalidate the client's H-cache when a write successfully commits. *Write-write serializability across blocks* is achieved by having the clients send in the older hash values at the time of the commit to detect concurrent write-sharing and having one or more of the writers to restart or redo the entire operation.

We emphasize here that, since *client state is mostly eliminated*, there is no need for a complicated recovery process or lease-based timeouts that are an inherent part of distributed locking-based approaches. Thus, our proposed scheme is inherently more robust and fault tolerant

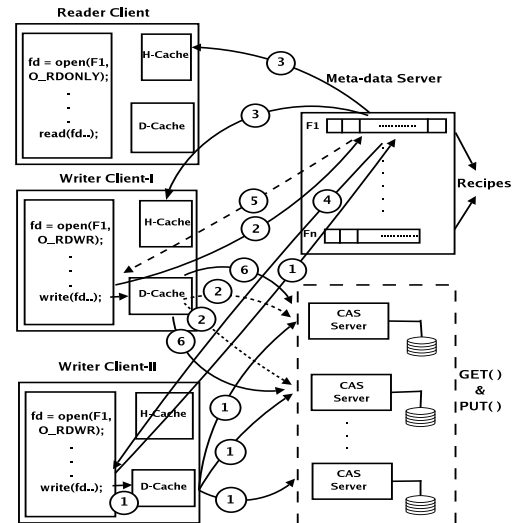


Figure 5: Action sequence: multiple-readers multiple-writers

from this perspective when H-caches are disabled. If H-caches are enabled however, temporary failures such as network disconnects can cause clients to read/write stale data. Further, the centralized meta-data server with no built-in support for replication is still a deterrent from the point of view of fault-tolerance and availability. We hope to address both these issues as future extensions.

3.6 Client-side Plug-in Architecture

The CAPFS design incorporates a client-side plug-in architecture that allows users to specify their own consistency policy to fine tune their application's performance. Figure 6 shows the hooks exported by the client-side and what callbacks a plug-in can register with the client-side daemon. Each plug-in is also associated with a "unique" name and identifier. The plug-in policy's name is used as a command-line option to the mount utility to indicate the desired consistency policy. The CAPFS client-side daemon loads default values based on the command-line specified policy name at mount time. The user is free to define any of the callbacks in the plug-ins (setting the remainder to NULL), and hence choosing the best trade-off between throughput and consistency for the application. The plug-in API/callbacks to be defined by the user provide a flexible and extensible way of defining a large range of (possibly non-standard) consistency policies. Additionally, other optimizations such as pre-fetching of data or hashes, delayed commits, periodic commits (e.g., commit after "t" units of time, or commit after every "n" requests), and others can be accommodated by the set of callbacks shown in Figure 6). For standard cases, we envision that the callbacks be used as follows.

Setting Parameters at Open: On mounting the CAPFS file system, the client-side daemon loads

<pre> struct plugin_policy_ops { handle (*pre_open)(force_commit, use_hcache, hcache_coherence, delay_commit, num_hashes); int (*post_open)(void *handle); int (*pre_close)(void *handle); int (*post_close)(void *handle); int (*pre_read)(void *handle, size, offset); int (*post_read)(void *handle, size, offset); int (*pre_write)(void *handle, size, offset, int *delay_wc); int (*post_write)(void *handle, sha_hashes *old, sha_hashes *new); int (*pre_sync)(const char *); int (*post_sync)(void *handle); }; </pre>	<pre> int hcache_get(void *handle, begin_chunk, nchunks, void *buf); int hcache_put(void *handle, begin_chunk, nchunks, const void *buf); int hcache_clear(void *handle); int hcache_clear_range(void *handle, begin_chunk, nchunks); void hcache_invalidate(void); int dcache_get(char *hash, void *buf, size); int dcache_put(char *hash, const void *buf, size); int commit(void *handle, sha_hashes *old_hashes, sha_hashes *new_hashes, sha_hashes *current_hashes); </pre>
Client-Side Plug-in API	CAPFS Client-Daemon: Core API

Figure 6: The client-side plug-in API and the CAPFS client-daemon core API. On receiving a system call, the CAPFS client-daemon calls the corresponding user-defined pre- and post- functions, respectively, before servicing the system call.

default values for `force_commit`, `use_hcache`, `hcache_coherence`, `delay_commit`, and `num_hashes` parameters. However, these values can be overridden on a per-file basis as well by providing a non-NULL `pre_open` callback. Section 3.4.4 indicates that in a commit operation, a client tells the server what it thinks the old hashes for the data are and then asks the server to replace them with new, locally calculated hashes. Hence a commit operation fails if the old hashes supplied by the client do not match the ones currently on the server (because of intervening commits by other clients). On setting the `force_commit` parameter, the client forces the server into accepting the locally computed hashes, overwriting whatever hashes the server currently has. The `use_hcache` parameter indicates whether the policy desires to use the H-Cache. The `hcache_coherence` parameter is a flag that indicates to the server the need for maintaining a coherent H-cache on all the clients that may have stale entries. The `delay_commit` indicates whether the commits due to writes should be delayed (buffered) at the client. The `num_hashes` parameter specifies how many hashes to fetch from the meta-data server at a time. These parameters can be changed by the user by defining a `pre_open` callback in the plug-in (Figure 6). This function returns a handle, which is cached by the client and is used as an identifier for the file. This handle is passed back to the user plug-in in `post_open` and other subsequent callbacks until the last reference to the file is closed. For instance, a plug-in implementing an AFS session like semantics [14] would fetch all hashes at the time of open, delay the commits till the time of a close, set the `force_commit` flag and commit all the hashes of a file at the end of the session.

Prefetching and Caching: Prior to a read, the client daemon invokes the `pre_read` callback (if registered). We envision that the user might desire to check H-Cache and D-Cache and fill them using the appropriate `hcache.get/dcache.get` API (Figure 6) exported by the client daemon. This callback might also be used to im-

plement prefetching data, hashes, and the like.

Delayed commits: A user might overload the `pre_write` callback routine to implement delayed commits over specific byte ranges. One possible way of doing this is to have the `pre_write` callback routine set a timer (in case a policy wishes to commit every “t” units of time) that would invoke the `post_write` on expiration. But for the moment, `pre_write` returns a value for `delay_wc` (Figure 6) to indicate to the core daemon that the write commits need to be delayed or committed immediately. Hence, on getting triggered, the `post_write` checks for pending commits and then initiates them by calling the appropriate core daemon API (`commit`). The `post_write` could also handle operations such as flushing or clearing the caches.

Summary: The callbacks provide enough flexibility to let the user choose when and how to implement most known optimizations (delayed writes, prefetching, caching, etc.) in addition to specifying any customized consistency policies. By passing in the offsets and sizes of the operations to the callback functions such as `pre_read`, `pre_write`, plug-in writers can also use more specialized policies at a very fine granularity (such as optimizations making use of MPI derived data-types [9]). This description details just one possible way of doing things. Users can use the API in a way that suits their workload, or fall back on standard predefined policies. Note that guaranteeing correctness of execution is the prerogative of the plug-in writer. Implementation of a few standard policies (Sequential, SESSION-like, NFS-like) and others (Table 1 in Section 4) indicate that this step does not place an undue burden on the user. The above plug-ins were implemented in less than 150 lines of C code.

4 Experimental Results

Our experimental evaluation of CAPFS was carried out on an IBM pSeries cluster. with the following configura-

tion. There are 20 compute nodes each of which is a dual hyper-threaded Xeon clocked at 2.8 GHz, equipped with 1.5 GB of RAM, a 36 GB SCSI disk and a 32-bit Myrinet card (LANai9.0 clocked at 134 MHz). The nodes run Redhat 9.0 with Linux 2.4.20-8 kernel compiled for SMP use and GM 1.6.5 used to drive the Myrinet cards. Our I/O configuration includes 16 CAS servers with one server doubling as both a meta-data server and a CAS server. All newly created files are striped with a stripe size of 16 KB and use the entire set of servers to store the file data. A modified version of MPICH 1.2.6 distributed by Myricom for GM was used in our experimental evaluations.

4.1 Aggregate Bandwidth Tests

Since the primary focus of parallel file systems is aggregate throughput, our first workload is a parallel MPI program (*pvfs_test.c* from the PVFS distribution), that determines the aggregate read/write bandwidths and verifies correctness of the run. The block sizes, iteration counts, and number of clients are varied in different runs. Consequently, this workload demonstrates concurrent-write sharing and sequential-write sharing patterns, albeit not simultaneously. Times for the read/write operations on each node are recorded over ten iterations and the maximum averaged time over all the tasks is used to compute the bandwidth achieved. The graphs for the above workload plot the aggregate bandwidth (in MB/s) on the y-axis against the total data transferred to or from the file system (measured in MB). The total data transferred is the product of the number of clients, block size and the number of iterations.

We compare the performance of CAPFS against a representative parallel file system – PVFS (Version 1.6.4). To evaluate the flexibility and fine-grained performance tuning made possible by CAPFS’ plug-in infrastructure, we divide our experimental evaluation of into categories summarized in Table 1. Five simple plug-ins have been implemented to demonstrate the performance spectrum.

The values of the parameters in Table 1 — (*force_commit*, *hcache_coherence* and *use_hcache*) dictate the consistency policies of the file system. The *force_commit* parameter indicates to the meta-data server that the commit operation needs to be carried out without checking for conflicts and being asked to retry. Consequently, this parameter influences write performance. Likewise, the *hcache_coherence* parameter indicates to the meta-data server that a commit operation needs to be carried out in strict accordance with the H-cache coherence protocol. Since the commit operation is not deemed complete until the H-cache coherence protocol finishes, any consistency policy that relaxes this requirement is also going to show performance improvements for writes. Note that neither of these two parameters is expected to have any significant effect on the read performance of this workload. On the other hand, using the

Policy Name	Use Hcache	Force Commit	Hcache Coherence
SEQ-1	0	0	X
SEQ-2	1	0	1
FOR-1	0	1	X
FOR-2	1	1	1
REL-1	1	1	0

Table 1: Design space constituting a sample set of consistency policies: SEQ-1, SEQ-2 implement sequential consistency; FOR-1, FOR-2 implement a slightly relaxed mechanism where commits are forced; REL-1 implements an even more relaxed mechanism. The X in rows 1 and 3 denotes a don’t care for the variable’s value.

H-cache on the client-side (*use_hcache* parameter) has the potential to improving the read performance because the number of RPC calls required to reach the data is effectively halved.

The first two rows of Table 1 illustrate two possible ways of implementing a sequentially consistent file system. The first approach denoted as SEQ-1, does not use the H-cache (and therefore H-caches need not be kept coherent) and does not force commits. The second approach denoted as SEQ-2, uses the H-cache, does not force commits, and requires that H-caches be kept coherent. Both approaches implement a sequentially consistent file system image and are expected to have different performance ramifications depending on the workload and the degree of sharing.

The third and fourth rows of Table 1 illustrate a slightly relaxed consistency policy where the commits are forced by clients instead of retrying on conflicts. The approach denoted as FOR-1, does not use the H-cache (no coherence required). The approach denoted as FOR-2, uses the H-cache and requires that they be kept coherent. One can envisage that such policies could be used in mixed-mode-environments where files are possibly accessed or modified by nonoverlapping MPI jobs as well as unrelated processes.

The fifth row of Table 1 illustrates an even more relaxed consistency policy denoted as REL-1, that forces commits, uses the H-cache, and does not require that the H-caches be kept coherent. Such a policy is expected to be used in environments where files are assumed to be non-shared among unrelated process or MPI-based applications or in scenarios where consistency is not desired. Note that it is the prerogative of the application-writer or plug-in developers to determine whether the usage of a consistency policy would violate the correctness of the application’s execution.

Read Bandwidth: In the case of the aggregate read bandwidth results (Figures 7(a) and 7(b)), the policies using the H-cache (SEQ-2, FOR-2, REL-1) start to perform better in comparison to both PVFS and policies not using the H-cache (SEQ-1, FOR-1). This tipping point occurs when the amount of data being transferred is fairly large (around 3 GB). This is intuitively correct, because the

larger the file, the greater the number of hashes that need to be obtained from the meta-data server. This requirement imposes a higher load on the server and leads to degraded performance for the uncached case. The sharp drop in the read bandwidth for the H-cache based policies (beyond 4 GB) is an implementation artifact caused by capping the maximum number of hashes that can be stored for a particular file in the H-cache.

On the other hand, reading a small file requires proportionately fewer hashes to be retrieved from the server, as well as fewer RPC call invocations to retrieve the entire set of hashes. In this scenario, the overhead of indexing and retrieving hashes from the H-cache is greater than the time it takes to fetch all the hashes from the server in one shot. This is responsible for the poor performance of the H-cache based policies for smaller file sizes. In fact, a consistency policy that utilizes the H-cache allows us to achieve a peak aggregate read bandwidth of about 450 MB/s with 16 clients. This is almost a 55% increase in peak aggregate read bandwidth in comparison to PVFS which achieves a peak aggregate read bandwidth of about 290 MB/s. For smaller numbers of clients, even the policies that do not make use of the H-cache perform better than PVFS.

In summary, for medium to large file transfers, from an aggregate read bandwidth perspective, consistency policies using the H-cache (SEQ-2, FOR-2, REL-1) outperform both PVFS and consistency policies that do not use the H-cache (SEQ-1, FOR-1).

Write Bandwidth: As explained in Section 3.3, write bandwidths on our system are expected to be lower than read bandwidths and these can be readily corroborated from Figures 7(c) and 7(d). We also see that PVFS performs better than all of our consistency policies for smaller data transfers (upto around 2 GB). At around the 1.5–2 GB size range, PVFS experiences a sharp drop in the write bandwidth because the data starts to be written out to disk on the I/O servers that are equipped with 1.5 GB physical memory. On the other hand no such drop is seen for CAPFS. The benchmark writes data initialized to a repeated sequence of known patterns. We surmise that CAPFS exploits this commonality in the data blocks, causing the content-addressable CAS servers to utilize the available physical memory more efficiently with fewer writes to the disk itself.

At larger values of data transfers (greater than 2 GB), the relaxed consistency policies that use the H-cache (REL-1, FOR-2) outperform both PVFS and the other consistency policies (SEQ-1, SEQ-2, FOR-1). This result is to be expected, because the relaxed consistency semantics avoid the expenses associated with having to retry commits on a conflict and the H-cache coherence protocol. Note that the REL-1 scheme outperforms the FOR-2 scheme as well, since it does not perform even the H-cache coherence protocol. Using the REL-1 scheme, we obtain a peak write bandwidth of about 320 MB/s with 16 clients, which is about a 12% increase in peak ag-

gregate write bandwidth in comparison to that of PVFS, which achieves a peak aggregate write bandwidth of about 280 MB/s.

These experiments confirm that performance is directly influenced by the choice of consistency policies. Choosing an overly strict consistency policy such as SEQ-1 for a workload that does not require sequential consistency impairs the possible performance benefits. For example, the write bandwidth obtained with SEQ-1 decreased by as much as 50% in comparison to REL-1. We also notice that read bandwidth can be improved by incorporating a client-side H-cache. For example, the read bandwidth obtained with SEQ-2 (FOR-2) increased by as much as 80% in comparison to SEQ-1 (FOR-1). However, this does not come for free, because the policy may require that the H-caches be kept coherent. Therefore, using a client-side H-cache may have a detrimental effect on the write bandwidth. All of these performance ramifications have to be carefully addressed by the application designers and plug-in writers before selecting a consistency policy.

4.2 Tiled I/O Benchmark

Tiled visualization codes are used to study the effectiveness of today's commodity-based graphics systems in creating parallel and distributed visualization tools. In this experiment, we use a version of the tiled visualization code [24] that uses multiple compute nodes, where each compute node takes high-resolution display frames and reads only the visualization data necessary for its own display.

We use nine compute nodes for our testing, which mimics the display size of the visualization application. The nine compute nodes are arranged in the 3 x 3 display as shown in Figure 8, each with a resolution of 1024 x 768 pixels with 24-bit color. In order to hide the merging of display edges, there is a 270-pixel horizontal overlap and a 128-pixel vertical overlap. Each frame has a file size of about 118 MB, and our experiment is set up to manipulate a set of 5 frames, for a total of about 600 MB.

This application can be set up to run both in collective I/O mode [9], wherein all the tasks of the application perform I/O collectively, and in noncollective I/O mode. Collective I/O refers to an MPI I/O optimization technique that enables each processor to do I/O on behalf of other processors if doing so improves the overall performance. The premise upon which collective I/O is based is that it is better to make large requests to the file system and cheaper to exchange data over the network than to transfer it over the I/O buses. Once again, we compare CAPFS against PVFS for the policies described earlier in Table 1. All of our results are the average of five runs.

Read Bandwidth: The aggregate read bandwidth plots (Figures 9(a) and 9(c)), indicate that CAPFS outperforms PVFS for both the noncollective and the col-

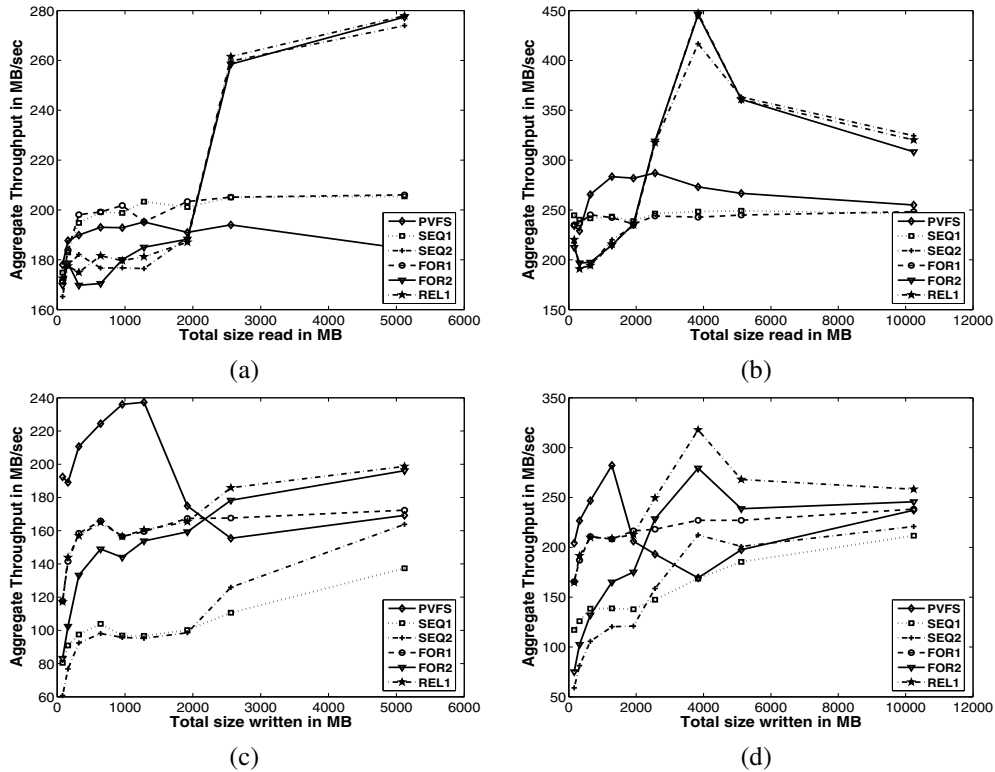


Figure 7: Aggregate Bandwidth in MB/s with varying block sizes: CAPFS vs. PVFS for (a) read-8-clients, (b) read-16-clients, (c) write-8-clients, (d) write-16-clients.

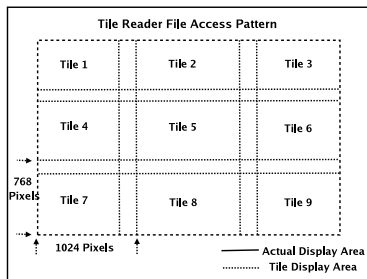


Figure 8: Tile reader file access pattern: Each processor reads data from a display file onto local display (also known as a tile).

lective I/O scenarios, across all the consistency policies. Note that the read phase of this application can benefit only if the policies use the H-caches (if available). As we saw in our previous bandwidth experiments, benefits of using the H-cache start to show up only for larger file sizes. Therefore, read bandwidths for policies that use the H-cache are not significantly different from those that don't in this application. Using our system, we achieve a maximum aggregate read bandwidth of about 90 MB/s without collective I/O and about 120 MB/s with collective I/O. These results translate to a performance improvement of 28% over PVFS read bandwidth for the noncollective scenario and 20% over PVFS read bandwidth for the collective scenario.

Write Bandwidth: The aggregate write bandwidths paint a different picture. For noncollective I/O, Figure 9

(b), the write bandwidth is very low for two of our policies (SEQ-2, FOR-2). The reason is that both these policies use an H-cache and also require that the H-caches be kept coherent. Also, the noncollective I/O version of this program makes a number of small write requests. Consequently, the number of H-cache coherence messages (invalidates) also increases, which in turn increases the time it takes for the writes to commit at the server. One must also bear in mind that commits to a file are serialized by the meta-data server and could end up penalizing other writers that are trying to write simultaneously to the same file. Note that the REL-1 policy does not lose out on write performance despite using the H-cache, since commits to the file do not execute the expensive H-cache coherence protocol. In summary, this result indicates that if a parallel workload performs a lot of small updates to a shared file, then any consistency policy that requires H-caches to be kept coherent is not appropriate from a performance perspective.

Figure 9(d) plots the write bandwidth for the collective I/O scenario. As stated earlier, since the collective I/O optimization makes large, well-structured requests to the file system, all the consistency policies (including the ones that require coherent H-caches) show a marked improvement in write bandwidth. Using our system, we achieve a maximum aggregate write bandwidth of about 35 MB/s without collective I/O and about 120 MB/s with

collective I/O. These results translate to a performance improvement of about 6% over PVFS write bandwidth for the noncollective scenario and about 13% improvement over PVFS write bandwidth for the collective scenario.

5 Related Work

Providing a plug-in architecture for allowing the user to *define* their own consistency policies for a parallel file system is a contribution unique to CAPFS file system. Tunable consistency models and tradeoffs with availability have been studied in the context of replicated services by Yu et al. [32].

Distributed file systems such as AFS [14], NFS [26] and Sprite [4, 20] have only a single server that doubles in functionality as a meta-data and data server. Because of the centralized nature of the servers, write atomicity is fairly easy to implement. Client-side caches still need to be kept consistent however, and it is with respect to this issue (write propagation) that these approaches differ from the CAPFS architecture. Coda [16] allows for server replication and it solves the write atomicity problem by having modifications propagated in parallel to all available replica servers (volume storages), and eventually to those that missed the updates.

Parallel file systems such as GPFS [27] and Lustre [6] employ distributed locking to synchronize parallel read-write disk accesses from multiple client nodes to its shared disks. The locking protocols are designed to allow maximum throughput, parallelism, and scalability, while simultaneously guaranteeing that file system consistency is maintained. Likewise, the Global File System (GFS) [22, 23] (a shared-disk, cluster file system) uses fine-grained SCSI locking commands, lock-caching and callbacks for performance and synchronization of accesses to shared disk blocks, and leases, journaling for handling node failures and replays. Although such algorithms can be highly tuned and efficient, failures of clients can significantly complicate the recovery process. Hence any locking-based consistency protocol needs additional distributed crash recovery algorithms or lease-based timeout mechanisms to guarantee correctness. The CAPFS file system eliminates much of the client state from the entire process, and hence client failures do not need any special handling.

Sprite-LFS [25] proposed a new technique for disk management, where all modifications to a file system are recorded sequentially in a log, which speeds crash recovery and writes. An important property in such a file system is that no disk block is ever overwritten (except after a disk block is reclaimed by the cleaner). Content-addressability helps the CAPFS file system gain this property, wherein updates from a process do not overwrite any existing disk or file system blocks. Recently, content-addressable storage paradigms have started to evolve that are based on distributed hash tables like

Chord [29]. A key property of such a storage system is that blocks are addressed by the cryptographic hashes of their contents, like SHA-1 [12]. Tolia et al. [30] propose a distributed file system CASPER that utilizes such a storage layer to opportunistically fetch blocks in low-bandwidth scenarios. Usage of cryptographic content hashes to represent files in file systems has been explored previously in the context of Single Instance Storage [5], Farsite [2], and many others. Similar to log-structured file systems, these storage systems share a similar no-overwrite property because every write of a file/disk block has a different cryptographic hash (assuming no collisions). CAPFS uses content-addressability in the hope of minimizing network traffic by exploiting commonality between data block, and to reduce synchronization overheads, by using hashes for cheap update based synchronization. The no-overwrite property that comes for free with content addressability has been exploited to provide extra concurrency at the data servers.

6 Concluding Remarks

In this paper, we have presented the design and implementation of a robust, high-performance parallel file system that offers user-defined consistency at a user-defined granularity using a client-side plug-in architecture. To the best of our knowledge CAPFS is the only file system that offers tunable consistency that is also user-defined and user-selectable at runtime. Rather than resorting to locking for enforcing serialization for read-write sharing or write-write sharing, CAPFS uses an optimistic concurrency control mechanism. Unlike previous network/parallel file system designs that impose a consistency policy on the users, our approach provides the mechanisms and defers the policy to application developers and plug-in writers.

7 Acknowledgments

We thank Robert Ross, Rajeev Thakur, and all the anonymous reviewers for their suggestions, which have greatly improved the content of this paper. This work was supported in part by Pittsburgh Digital Greenhouse, NSF ITR-0325056 and CCR-0130143.

Notes

¹ Work done while at Penn State University. Current Address: Mathematics and Computer Science Division, Argonne National Laboratory, IL 60439; vilayann@mcs.anl.gov.

References

- [1] Technical Report and CAPFS Source code. <http://www.cse.psu.edu/~vilayann/capfs/>.
- [2] A. Adya, W. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer.

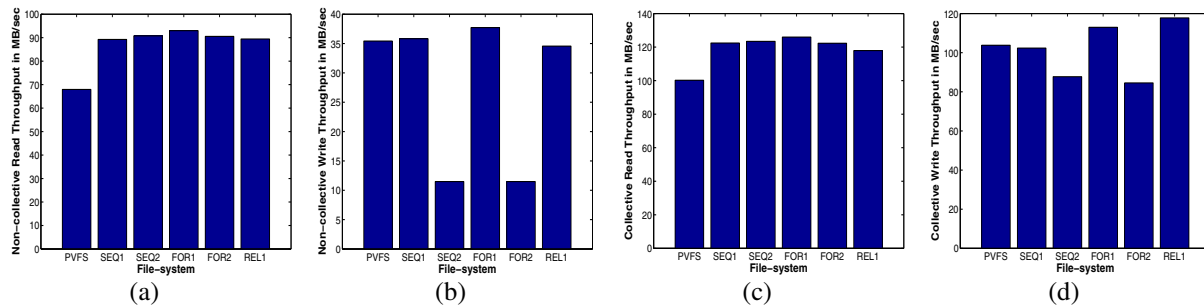


Figure 9: File I/O benchmark bandwidth in MB/s: (a) noncollective read, (b) noncollective write, (c) collective read, (d) collective write.

- FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.
- [3] M. B. Alexander. *Assessment of Cache Coherence Protocols in Shared-Memory*. PhD thesis, University of Toronto, 2003.
 - [4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery SIGOPS, 1991.
 - [5] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, August 2000.
 - [6] P. J. Braam. The Lustre Storage Architecture, <http://www.lustre.org/documentation.html>, August 2004.
 - [7] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.
 - [8] P. F. Corbett and D. G. Feitelson. The Vesta Parallel File System. In *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE Computer Society Press and Wiley, 2001.
 - [9] M. P. I. Forum. MPI-2: Extensions to the Message-Passing Interface, 1997. <http://www.mpi-forum.org/docs>.
 - [10] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, 2004.
 - [11] C. Gray and D. Cheriton. Leases: An Efficient Fault-tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.
 - [12] N. W. Group. RFC 3174 - US Secure Hash Algorithm - I (SHA1), 2001.
 - [13] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Language Systems*, 15(5), 1993.
 - [14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), 1988.
 - [15] IBM. Distributed Lock Manager for Linux, 2001. <http://oss.software.ibm.com/dlm>.
 - [16] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991.
 - [17] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2), 1981.
 - [18] C. May, E. Silha, R. Simpson, and H. Warren. The PowerPC Architecture: A Specification for a New Family of RISC Processors, 1994. Morgan Kaufman, San Francisco, CA, Second Edition.
 - [19] S. J. Mullender and A. S. Tanenbaum. A Distributed File Service based on Optimistic Concurrency Control. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1985.
 - [20] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*. ACM Press, 1987.
 - [21] B. D. Noble and M. Satyanarayanan. An Empirical Study of a Highly Available File System. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and Modeling of Computer Systems*, 1994.
 - [22] K. W. Preslan and A. P. Barry. A 64-bit, Shared Disk File System for Linux. Technical report, Sistina Software, Inc, 1999.
 - [23] K. W. Preslan, A. P. Barry, J. Brassow, R. Cattelan, A. Manthei, E. Nygaard, S. V. Oort, D. Teigland, M. Tilstra, M. O'Keefe, G. Erickson, and M. Agarwal. Implementing Journaling in a Linux Shared Disk File System. In *IEEE Symposium on Mass Storage Systems*, 2000.
 - [24] R. B. Ross. Parallel I/O Benchmarking Consortium, <http://www-unix.mcs.anl.gov/~rross/pio-benchmark>.
 - [25] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1), 1992.
 - [26] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, 1985.
 - [27] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the First Conference on File and Storage Technologies (FAST)*, 2002.
 - [28] I. Standard. 1003.1 Portable Operating System Interface (POSIX)-Part 1: System Application Program Interface (API) [C Language], 1996.
 - [29] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1), 2003.
 - [30] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic Use of Content Addressable Storage for Distributed File Systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, 2003.
 - [31] M. Uysal, A. Acharya, and J. Saltz. Requirements of I/O Systems for Parallel Machines: An Application-driven Study. Technical Report CS-TR-3802, University of Maryland, College Park, MD, 1997.
 - [32] H. Yu. TACT: Tunable Availability and Consistency Tradeoffs for Replicated Internet Services (poster session). *SIGOPS Operating Systems Review*, 2000.

MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices

Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, Walid A. Najjar
Dept. of Computer Science Dept. of Computer Science & Engineering
University of Cyprus University of California, Riverside
dzeina@cs.ucy.ac.cy {slin,vana,dg,najjar}@cs.ucr.edu

Abstract

In this paper we propose the *MicroHash* index, which is an efficient external memory structure for *Wireless Sensor Devices* (WSDs). The most prevalent storage medium for WSDs is *flash memory*. Our index structure exploits the asymmetric read/write and wear characteristics of flash memory in order to offer high performance indexing and searching capabilities in the presence of a low energy budget which is typical for the devices under discussion. A key idea behind *MicroHash* is to eliminate expensive random access deletions. We have implemented *MicroHash* in nesC, the programming language of the TinyOS [7] operating system. Our trace-driven experimentation with several real datasets reveals that our index structure offers excellent search performance at a small cost of constructing and maintaining the index.

1 Introduction

The improvements in hardware design along with the wide availability of economically viable embedded sensor systems enable researchers nowadays to sense environmental conditions at extremely high resolutions. Traditional approaches to monitor the physical world include passive sensing devices which transmit their readings to more powerful processing units for storage and analysis. *Wireless Sensor Devices* (WSDs) on the other hand, are tiny computers on a chip that is often as small as a coin or a credit card. These devices feature a low frequency processor ($\approx 4\text{--}58\text{MHz}$) which significantly reduces power consumption, a small on-chip flash memory ($\approx 32\text{KB--}512\text{KB}$) which can be used as a temporary local storage medium, a wireless radio for communication, on-chip sensors, and an energy source such as a set of AA batteries or solar panels [13]. This multitude of features constitute WSDs powerful devices which can be used for in-network processing, filtering and aggregation [11, 12, 16]. Large-scale deployments of sensor

network devices have already emerged in environmental and habitat monitoring [13, 15], seismic and structural monitoring [24], factory and process automation and a large array of other applications [11, 12, 16].

In long-term deployments, it is often cheaper to keep a large window of measurements in-situ (at the generating site) and transmit the respective information to the user only when requested (this is demonstrated in Section 2.4). For example, biologists analyzing a forest are usually interested in the long-term behavior of the environment. Therefore the sensors are not required to transmit their readings to a *sink* (querying node) at all times. Instead, the sensors can work unattended and store their reading locally until certain preconditions are met, or when the sensors receive a query over the radio that requests the respective data. Such in-network storage conserves energy from unnecessary radio transmissions, which can be used to increase the sampling frequency of the data and hence the fidelity of the measurements in reproducing the actual physical phenomena and prolong the lifetime of the network.

Currently, the deployment of the sensor technology is severely hampered by the lack of efficient infrastructure to store locally large amounts of sensor data measurements. The problem is that the local RAM memory of the sensor nodes is both volatile and very limited ($\approx 2\text{KB--}64\text{KB}$). In addition, the non-volatile on-chip flash memory featured by most sensors is also very limited ($\approx 32\text{KB--}512\text{KB}$). However the limited local storage of sensor devices is expected to change soon. Several sensor devices, such as the RISE [1] hardware platform, include off-chip flash memory which supplements each sensor with several megabytes of storage. Flash memory has a number of distinct characteristics compared to other storage media: First, each page (typically 128B–512B) can only be written a limited number of times ($\approx 10,000\text{--}100,000$). Second, pages can only be written after they have been deleted in their entirety. However, a page deletion always triggers the deletion of its respec-

tive block ($\approx 8\text{KB}$ - 64KB per block). Due to these fundamental constraints, efficient storage management becomes a challenging task.

The problem that we investigate in this paper is how to efficiently organize the data locally on flash memory. Our desiderata are:

1. To provide efficient access to the data stored on flash by *time* or *value*, for *equality* queries generated by the user.
2. To increase the *longevity* of the flash memory by spreading page writes out uniformly so that the available storage capacity does not diminish at particular regions of the flash media.

We propose the *MicroHash* index, which serves as a primitive structure for efficiently indexing temporal data and for executing a wide spectrum of queries. Note that the data generated by sensor nodes has two unique characteristics: i) Records are generated at a given point in time (i.e. these are temporal records), and ii) The recorded readings are numeric values in a limited range. For example a temperature sensor might only record values between -40°F to 250°F with one decimal point precision. Traditional indexing methods used in relational database systems are not suitable as these do not take into account the asymmetric read/write behavior of flash media. Our indexing techniques have been designed for sensor nodes that feature large flash memories, such as the RISE [1] sensor, which provide them with several MBs of storage. MicroHash has been implemented in nesC [6] and uses the TinyOS [7] operating system.

In this paper we make the following contributions:

1. We propose the design and implementation of MicroHash, a novel index structure for supporting equality queries in sensor nodes with limited processing capabilities and a low energy budget.
2. We present efficient algorithms for inserting, deleting and searching data records stored on flash.
3. We describe the prototype implementation of MicroHash in nesC [6], and demonstrate the efficiency of our approach with an extensive experimental study using atmospheric readings from the University of Washington [21] and the Great Duck Island study [15].

2 The Memory Hierarchy

In this section we briefly overview the architecture of a sensor node, with a special focus on its memory hierarchy. We also study the distinct characteristics of flash memory and address the challenges with regards to energy consumption and access time.

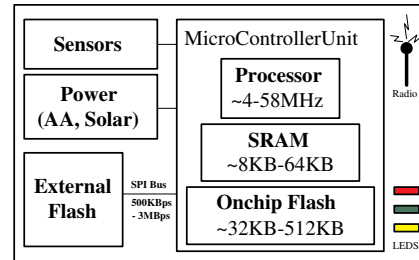


Figure 1: The Architecture of a typical Wireless Sensor.

2.1 System Architecture

The architecture of a sensor node (see Figure 1), consists of a microcontroller unit (MCU) which is interconnected to the radio, the sensors, a power source and the LEDs. The MCU includes a processor, a static RAM (SRAM) module and an on-chip flash memory. The processor runs at low frequencies (≈ 4 - 58MHz) which reduces power consumption. The SRAM is mainly used for code execution while in the latest generation of sensors, such as Yale's 58MHz XYZ node [10] and the Intel's 12MHz iMote (<http://www.intel.com>), it can also be used for *in-memory* (or SRAM) buffering. The choice of the right energy source is application specific. Most sensors either deploy a set of AA batteries or solar panels [13]. Therefore a sensor node might have a very long lifetime.

The on-chip flash provides a small non-volatile storage area (32KB - 512KB) for storing the executable code or for accumulating values for a small window of time [11]. A larger external storage can also be supplemented to a sensor using the *Serial Peripheral Interface (SPI)* which is typically found on these devices. For example in the RISE platform, nodes feature a larger off-chip flash memory which provides the sensor with several MBs of storage. The external flash memory is connected to the MCU through a Serial Peripheral Interface (SPI), that operates at a fraction of the CPU frequency (e.g. $\frac{\text{cpu freq}}{8}$). Therefore a faster processor would increase the maximum throughput of the SPI interface.

Although it is currently not clear whether Moore's Law will apply to the size and price of the sensor units or their hardware characteristics, we believe that future sensor nodes will feature more SRAM and flash storage, as more complex in-network processing applications, increase the memory and potentially the CPU demand.

2.2 Overview of Flash Memory

Flash Memory is the most prevalent storage media used in current sensor systems because of its many advantages including: i) non-volatile storage, ii) simple cell architecture, which allows easy and economical production,

iii) shock-resistance, iv) fast read access and power efficiency. These characteristics establish flash memory as an ideal storage media for mobile and wireless devices [3].

There are two different types of flash memory, *NOR flash* and *NAND flash*, which are named according to the logic gate of their respective storage cell. NAND flash is the newer generation of flash memory which is characterized by faster erase time, higher durability and higher density. NOR is an older type of flash which is mainly used for code storage (e.g. for the BIOS). Its main advantage is that it supports writes at a byte granularity as opposed to page granularity used in NAND flash. NOR flash has also faster access time (i.e. $\approx 200\text{ns}$) than NAND ($50\text{--}80\mu\text{s}$) but lacks in all other characteristics such as density and power efficiency.

For the rest of the paper we will focus on the characteristics of NAND memory as this is the type of memory used for the on-chip and off-chip flash of most sensors including the RISE platform. According to Micron (<http://www.micron.com/>), NAND memory is the fastest growing memory market in 2005 (\$8.7 billion). NAND flash features a number of distinct constraints which can be summarized as following:

1. **Read-Constraint:** Reading data stored on flash memory can be performed at granularities ranging from a single byte to a whole block (typically 8KB–64KB).
2. **Delete-Constraint:** Deleting data stored on flash memory can only be performed at a block granularity (i.e. 8KB–64KB).
3. **Write-Constraint:** Writing data can only be performed at a page granularity (typically 256B–512B), after the respective page (and its respective 8KB–64KB block) has been deleted.
4. **Wear-Constraint:** Each page can only be written a limited number of times (typically 10,000–100,000).

The design of our MicroHash index structure in Section 5, considers the above constraints.

2.3 Access Time of NAND Flash

Table 1, presents the average measurements that we obtained from a series of micro-benchmarks using the *RISE* platform along with a HP E3630A constant 3.3V power supply and a Fluke 112 RMS Multimeter. The first observation is that reading is three orders of magnitude less power demanding than writing. On the other hand, block erases are also quite expensive but can be performed much faster than the former two operations. Note that

NAND Flash installed on a Sensor Node			
	Page Read 1.17mA	Page Write 37mA	Block Erase 57mA
Time	6.25ms	6.25ms	2.26ms
Data Rate	82KBps	82KBps	7MBps
Energy	24 μ J	763 μ J	425 μ J
	Page Erase-Write 43mA	Flash Idle 0.068mA	Flash Sleep 0.031mA
Time	6.75ms	N/A	N/A
Data Rate	76KBps	N/A	N/A
Energy	957 μ J	220 μ J/sec	100 μ J/sec

Table 1: Performance Parameters for NAND Flash using a 3.3V voltage, 512B Page size and 16KB Block size

read and write operations involve the transfer of data between the MCU and the SPI bus, which becomes the bottleneck in the time to complete the operation. Specifically, reading and writing on flash media without the utilization of the SPI bus can be achieved in $\approx 50\mu$ and $\approx 200\mu\text{s}$ respectively [22]. Finally, our results are comparable to measurements reported for the MICA2 mote in [2] and the XYZ sensor in [10].

Although these are hardware details, the application logic needs to be aware of these characteristics in order to minimize energy consumption and maximize performance. For example, the deletion of a 512B page will trigger the deletion of a 16KB block on the flash memory. Additionally the MCU has to re-write the rest unaffected 15.5KB. One of the objectives of our index design is to provide an abstraction which hides these hardware specific details from the application.

2.4 Energy Consumption of NAND Flash

Another question is whether it is cheaper to write to flash memory rather than transmitting over the RF radio. We used the RISE mote to measure the cost of transmitting the data over a 9.6Kbps radio (at 60mA), and found that transmitting 512B (one page) takes on average 416ms or 82,368 μ J. Comparing this with the 763 μ J required for writing the same amount of data to local flash, along with the fact that transmission of one byte is roughly equivalent to executing 1120 CPU instructions, makes local storage and processing highly desirable.

A final question we investigated is how many bytes we can store on local flash before a sensor runs out of energy. Note that this applies only to the case where the sensor runs on batteries. Double batteries (AA) used in many current designs operate at a 3V voltage and supply a current of 2500 mAh (milliAmp-hours). Assuming similarly to [15], that only 2200mAh is available and that all current is used for data logging, we can calcu-

late that AA batteries offer 23,760J (2200mAh * 60 * 60 * 3). With a 16KB block size and a 512B page size, we would have one block delete every 32 page writes (16KB/512B). Writing a page, according to our measurements, requires 763μJ while the cost of performing a block erase is 425μJ. Therefore writing 16KB requires:

$$\begin{aligned} Write_{16KB} &= \underbrace{(32 \text{ pages} * 763\mu J)}_{\text{write cost}} + \underbrace{(425\mu J)}_{\text{block-erase cost}} \\ &= 24,841\mu J \end{aligned}$$

Using the result from the above equation, we can derive that by utilizing the 23,760J offered by the batteries, we can write ≈15GB before running out of batteries ((23,760J * 16KB) / 24,841μJ). An interesting point is that even in the absence of a wear-leveling mechanism we would be able to accommodate the 15GB without exhausting the flash media. However this would not be true if we used solar panels [13], which provide a virtually unlimited power source for each sensor device. Another reason why we want to extend the lifetime of the flash media is that the batteries of a sensor node could be replaced in cases where the devices remain accessible.

3 Problem Definition

In this section we provide a formal definition of the indexing problem that the MicroHash index addresses. We also describe the structure of the MicroHash index and explain how it copes with the distinct characteristics of flash memory.

Let S denote some sensor that acquires readings from its environment every ϵ seconds (i.e. $t = 0, \epsilon, 2\epsilon, \dots$). At each time instance t , the sensor S obtains a temporal data record $drec = \{t, v_1, v_2, \dots, v_x\}$, where t denotes the timestamp (key) on which the tuple was recorded, while v_i ($1 \leq i \leq x$) represents the value of some reading (such as humidity, temperature, light and others).

Also let $P = \{p_1, p_2, \dots, p_n\}$ denote a flash media with n available pages. A page can store a finite number of bytes (denoted as p_i^{size}), which limits the capacity of P to $\sum_{i=0}^n p_i^{size}$. Pages are logically organized in b blocks $\{block_1, block_2, \dots, block_b\}$, each block containing n/b consecutive pages. We assume that pages are read on a page-at-a-time basis and that each page p_i can only be deleted if its respective block (denoted as p_i^{block}) is deleted as well (write/delete-constraint). Finally due to the wear-constraint, each page can only be written a limited number of times (denoted as p_i^{wc}).

The MicroHash index supports efficient *value-based equality queries* and efficient *time-based equality* and *range queries*. These queries are defined as follows:

Definition 1. Value-Based Equality Queries: A *One dimensional query* $Q(v_i, a)$ in which the field values of attribute v_i are equivalent to value a .

For example the query $q=(temperature, 95F)$ can be used to find time instances (ts) and other recorded readings when the temperature was 95F.

Definition 2. Time-Based Range and Equality Queries: A *range query* is a *one dimensional query* $Q(t, a, b)$ in which the time attribute t , is between the lower and upper bound a and b respectively. The *equality query* is a special case of the range query $Q(t, a, b)$ in which $a = b$.

For example the query $q=(ts, 100, 110)$ can be used to find the tuples recorded in the 10 second interval.

Evaluating the above queries efficiently requires that the system maintains an index structure along with the generated data. Specifically, while a node senses data from its environment (i.e. data records), it also creates index entries that point to the respective data stored on the flash media. When a node needs to evaluate some query, it uses the index records to quickly locate the desired data. Since the number of index records might be potentially very large, these are stored on the external flash as well. Although maintaining index structures is a well studied problem in the database community [4, 9, 19], the low energy budget of sensor nodes along with the unique read, write, delete and wear constraints of flash memory introduce many new challenges. In order to maximize efficiency our design objectives are as follows:

1. **Wear-Leveling:** Spread page writes out uniformly across the storage media P in order to avoid wearing out specific pages.
2. **Block-Erase:** Minimize the number of *random-access deletions* as the deletion of individual pages triggers the deletion of the whole respective block.
3. **Fast-Initialization:** Minimize the size of in-memory (SRAM) structures that will be required in order to use the index.

4 MicroHash Data Structures

In this section we describe the data structures created in the fast but volatile SRAM to provide an efficient way to access data stored on the persistent but slower flash memory. First we describe the underlying organization of data on the flash media and then describe the involved in-memory data structures.

4.1 Flash Organization

MicroHash uses a Heap Organization, in which records are stored on the flash media in a circular array fashion.

This allows data records to be naturally sorted based on their timestamp and therefore our organization is *Sorted by Timestamp*. This organization requires the least overhead in SRAM (i.e. only one data write-out page). Additionally, as we will show in Section 5.4, this organization addresses directly the delete, write and wear constraint. When the flash media is full we simply delete the next block following *idx*. Although other organizations in relational database systems, such as *Sorted* or *Hashed* on some attribute could also be used, they would have a prohibitive cost as the sensor would need to continuously update written pages (i.e. perform an expensive random page write). On the other hand, our Heap Flash Organization always yields completely full data pages as data records are consecutively packed on the flash media.

4.2 In-Memory (SRAM) Data Structures

The flash media is segmented into n pages, each with a size of 512B. Each page consists of a 8B **header** and a 504B **payload**.

Specifically the **header** includes the following fields (also illustrated in Figure 2):

- i) A 3-bit *Page Type (TYP)* identifier, used to for the different types of pages (data, index, directory and root).
- ii) A 16-bit *Cyclic Redundancy Checking (CRC)* polynomial on the payload, which can be used for integrity checking.
- iii) A 7-bit *Number of Records (SIZ)*, which identifies how many records are stored inside a page. We use fixed size records because records generated by a sensor always have the same size.
- iv) A 23-bit *Previous Page Address (PPA)*, stores the address of some other page on the flash media giving in that way the capability to create linked lists on the flash.
- v) A 15-bit *Page Write Counter (PWC)*, which keeps the number of times a page has been written to flash.

While the header is identical for any type of page, the **payload** can store four different types of information:

- i) *Root Page*: contains information related to the state of the flash media. For example it contains the position of the last write (*idx*), the current cycle (*cycle*) and meta-information about the various indexes stored on the flash media.
- ii) *Directory Page*: contains a number of directory records (buckets) each of which contains the address of the last known index page mapped to this bucket. In order to form larger directories several directory pages might be chained using the 23-bit PPA address in the header.
- iii) *Index Page*: contains a fixed number of index records and the 8 byte timestamp of the last known data record. The latter field, denoted as *anchor* is exploited by timestamp searches which can make an informed decision on which page to follow next. Additionally, we evaluate two alternative index record layouts. The first, denoted as *offset* layout, maintains for each data record

```
typedef struct Page {
    uint8_t typ:3; // optional anchor
    uint16_t crc:16;
    uint16_t pwc:15;
    uint8_t siz:7;
    uint32_t ppa:23;
    union {
        RootP rootP;
        DirP dirP;
        IdxP idxP;
        DataP dataP;
    };
} __attribute__((packed));

typedef struct IdxP {
    // optional anchor
    uint64_t lastTS;
    IdxRec records[IREC];
} __attribute__((packed));

typedef struct DataRec {
    timestamp_t ts;
    data_t val;
} __attribute__((packed));

typedef struct IdxRec {
    fladdress_t datap;
    DataRec records[DREC]; // optional offset
} __attribute__((packed));

__attribute__((packed));
```

Figure 2: Main data structures used in our nesC implementation of the MicroHash Index.

a respective *pageid* and *offset*, while the second layout, denoted as *nooffset*, maintains only the *pageid* of the respective data record.

- iv) *Data Page*: contains a fixed number of data records. For example when the record size is 16B then each page can contain 31 consecutively packed records.

5 Indexing in MicroHash

The *MicroHash* index is an efficient external-memory structure designed to support equality queries in sensor nodes that have limited main memory and processing capabilities. A *MicroHash* index structure consists of two substructures: i) A *Directory* and ii) a set of *Index Pages*. The *Directory* consists of a set of buckets. Each bucket maintains the address of the newest (chronologically) index page that maps to that bucket. The *Index Pages* contain the addresses of the data records that map to the respective bucket. Note that there might be an arbitrarily large number of data and the index pages. Therefore these pages are stored on the flash media and fetched into main memory only when requested.

The *MicroHash* index is built while data is being acquired from the environment and stored on the flash media. In order to better describe our algorithm we divide its operation in four conceptual phases: *a) The Initialization Phase* in which the root page and certain parts of the directory are loaded into SRAM, *b) The Growing Phase* in which data and index pages are sequentially inserted and organized on the flash media, *c) The Repartition Phase* in which the index directory is re-organized such that only the directory buckets with the highest hit ratio remain in memory, and the *d) The Deletion Phase* which is triggered for garbage collection purposes.

5.1 The Initialization Phase

In the first phase the MicroHash index locates the root page on flash media. In our current design, the root page is written on a specific page on flash (page0). If page0 is worn out, we recursively use the next available page. Therefore a few blocks are pre-allocated at the beginning of the flash media for the storage of root pages. The root

page indicates what type of indexes are available on the system and the addresses of their respective directories. Given that an application requires the utilization of an index I , the system pre-loads part of I 's directory into SRAM (detailed discussion follows in Section 5.3). The root and directory pages then remain in SRAM, for efficiency, and are periodically written out to flash.

5.2 The Growing Phase

Let us assume that a sensor generates a temporal record $drec = \{t, v_1, v_2, \dots, v_x\}$ every ϵ seconds, where t is the timestamp on which the record was generated and v_i ($1 \leq i \leq x$) some distinct reading (e.g. humidity, temperature, etc). Instead of writing $drec$ directly to flash, we use an in-memory (SRAM) buffer page p^{write} . When p^{write} gets full it is flushed to the address idx , where idx denotes the address after the last page write. Note that idx starts out as zero and this counter is incremented by one every time a page is written out. When idx becomes equal to the size of the flash media n , it is reset to zero. In order to provide a mechanism for finding the relative chronological order of pages written on the flash media, we also maintain the counter $cycle$, which is incremented by one every time idx is reset to zero. The combination of the $\langle cycle, pageid \rangle$ provides this mechanism.

Next we describe how index records are generated and stored on the flash media. The index records in our structure are generated whenever the p^{write} gets full. At this point we can safely determine the physical address of the records in p^{write} (i.e. idx). We create one index record $ir = [idx, offset]$ for each data record in p^{write} ($\forall drec \in p^{write}$). For example assume that we insert the following 12 byte $[timestamp, value]$ records into an empty *MicroHash* index: $\{[1000, 50], [1001, 52], [1002, 52]\}$. This will trigger the creation of the following index records: $\{[0, 0], [0, 12], [0, 24]\}$. Since p^{write} is written to address idx the index records always reference data records that have a smaller $\langle cycle, pageid \rangle$ identifier.

The *MicroHash Directory* provides the start address of the index pages. It is constructed by providing the following three parameters: a) A lower bound (lb) on the indexed attribute, b) an upper bound (ub) on the indexed attribute and the number of available buckets c (note that we can only fit a certain number of directory buckets in memory). For example assume that we index temperature readings which are only collected in the following known and discrete range $[-40..250]$, then we set $lb = -40F$, $ub = 250F$ and $c = 100$. Initially each bucket represents exactly $\frac{[lb..ub]}{c}$ consecutive values although this equal splitting (which we call *equiwidth splitting*) is refined in the repartition phase based on the data values collected at run-time.

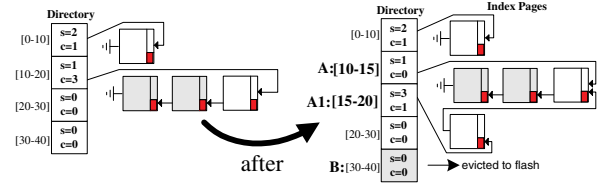


Figure 3: The Repartition Phase.

5.3 The Repartition Phase

A drawback of the initial *equiwidth bucket* splitting approach is that some buckets may rarely be used while others may create long lists of index records. To overcome this problem, we use the following splitting policy: Whenever a directory bucket A links to more than τ records (user parameter), we evict to flash the bucket B , which was not used for the longest period of time. Note that this mechanism can be implemented using only two counters per bucket (one for the timestamp and one for the number of records). In addition to the eviction of page B , we also create a new bucket $A1$. Our objective is to provide a finer granularity to the entries in A as this bucket is the most congested one. Note that the values in A are not reassigned between A and $A1$ as it would happen in dynamic hashing techniques, such as *extendible hashing* [4] or *linear hashing* [9]. The reason is that the index pages are on the flash media and updating these pages would result in a potentially very large number of random updates (which would be extremely expensive). Our *equidepth*, rather than *equiwidth*, bucket splitting approach keeps in memory finer intervals for index records used more frequently.

Figure 3 shows that each bucket is associated with a counter s , that indicates the timestamp of the last time the buffer was used, and a counter c that indicates the number of index records added since the last split. In the example, the $c = 3$ value in bucket 2 ($A:[10-20]$) exceeds the $\tau = 2$ threshold and therefore the index forces bucket 4 ($B:[30-40]$) to the flash media while bucket two is split into $A:[10-15]$ and $A1:[15-20]$. Note that the A list now contains values in $[10-20]$ while the $A1$ list contains only values in the range $[15-20]$.

5.4 The Deletion Phase

In this phase the index performs a garbage collection operation of the flash media in order to make space for new data. The phase is triggered after all n pages have been written to the flash media. This operation blindly deletes the next n/b pages (the whole next block starting at idx). It is then triggered again whenever n/b pages have been written, where b is the number of blocks on the flash media. That leaves the index with n/b clean pages that

can be used for future writes. Note that this might leave pointers from index pages referencing data that is already deleted. This problem is handled by our search algorithm described in the next section.

The distinct characteristic of our garbage collection operation is that it satisfies directly the delete-constraint, because pages are deleted in blocks (which is cheaper than deleting a page-at-a-time). This makes it different from similar operations of flash file systems [2, 17] that perform page-at-a-time deletions. Additionally, this mode provides the capability to "blindly" delete the next block without the need to read or relocate any of the deleted data. The correctness of this operation is established by the fact that the index records always reference data records that have a smaller $\langle \text{cycle}, \text{pageid} \rangle$ identifier. Therefore when an index page is deleted then we are sure that all associated data pages are already deleted.

6 Searching in MicroHash

In this section we show how records can efficiently be located by their value or timestamp.

6.1 Searching by Value

The first problem we consider is how to perform *value-based equality queries*. Finding records by their value involves: a) locating the appropriate directory bucket, from which the system can extract the address of the last index page, b) reading the respective index pages on a page-by-page basis and c) reading the data records referred by the index pages on a page-by-page basis. Since SRAM is extremely limited on a sensor node we adopt a *record-at-a-time* query return mechanism, in which records are reported to the caller on record-by-record basis. This mode of operation requires three available pages in SRAM, one for the directory (dirP) and two for the reading (idxP, dataP), which only occupies 1.5KB. If more SRAM was available, the results could have been returned at other granularities as well. The complete search procedure is summarized in Algorithm 1.

Note that the loadPage procedure in line 4 and 6 returns NULL if the fetched page is not in valid chronological order (with respect to its preceding page) or, if the data records, in data pages, are not within the specified bucket range. This is consequence of the way the garbage collector operates, as it does not update the index records during deletions for performance reasons. However, these simple checks applied by loadPage ensure that we can safely terminate the search at this point. Finally, since the MicroHash index returns records on a record-at-a-time basis, we use a final *signal finished* which notifies the application that the search procedure has been completed.

Algorithm 1 EqualitySearch

Input: *value*: the query (search predicate).

Output: The records that contains *value*.

```

1: procedure EQUALITYSEARCH(value)
2:   bucket = hash(value);
3:   address = dirP[bucket].idxP;
4:   while ((idxP = loadPage(address)) != NULL) do
5:     for i = 0 to |idxP.size| do
6:       If ((dataP = loadPage(idxP[i].dataP)) == NULL)
7:         address = 0; break;
8:       If (dataP.record[idxP[i].offset] == value)
9:         signal dataP.record[idxP[i].offset];
10:    end for
11:    address = idxP.ppp;
12:  end while
13:  signal finished;
14: end procedure

```

6.2 Searching by Timestamp

In this section we investigate *time-based equality and range queries*. First, note that if index pages were stored in a separate physical location, and thus not interleaved with data pages, the sorted (by timestamp) file organization would allow us to access any data record in $O(1)$ time. However, this would also violate our wear leveling mechanism as we wouldn't be able to spread out the page writes uniformly among data and index pages. Another approach would be to deploy an in-memory address translation table, such as the one used in [22] and [23], which would hide the details of wear-leveling mechanism. However, such a structure might be too big given the memory constraints of a sensor node and would also delay the sensor boot time.

Efficient search can be supported by a number of different techniques. One popular technique is to perform a binary search over all pages stored on the flash media. This would allow us to search in $O(\log n)$ time, where n is the size of the media. However, for large values of n such a strategy is still expensive. For example with a 512MB flash media and a page size of 512B we would need approximately 20 page reads before we find the expected record.

In our approach we investigate two binary search variants named: *LBSearch* and *ScaleSearch*. *LBSearch* starts out by setting a pessimistic lower bound on which page to examine next, and then recursively refines the lower bound until the requested page is found. *ScaleSearch* on the other hand exploits knowledge about the underlying distribution of data and index pages in order to offer a more aggressive search method that usually executes faster. *ScaleSearch* is superior to *LBSearch* when data and index pages are roughly uniformly distributed on the flash media but its performance deteriorates for skewed distributions.

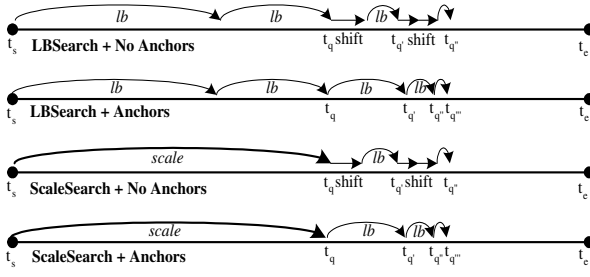


Figure 4: Searching By Timestamp. t_s : oldest timestamp on flash (t_e : newest), t_q : the query (timestamp), lb : The lower bound obtained using either idx_{lb} or idx_{scaled} .

For the remainder of this section we assume that a sensor S maintains locally some indexed readings for the interval $[t_a..t_b]$. Also let $x < y$ (and $x > y$) denote that the $\langle cycle_x, idx_x \rangle$ pair of x is smaller (and respectively greater) than the $\langle cycle_y, idx_y \rangle$ of y . When S is asked for a record with the timestamp t_q , it follows one of the following approaches:

i) *LBSearch*: S starts out by setting the lower bound :

$$idx_{lb}(t_q, t_s) = \begin{cases} \left\lceil \frac{t_q - t_s}{\mathfrak{R}} \right\rceil, & \text{if cycle} == 0; \\ idx + \left\lceil \frac{t_q - t_s}{\mathfrak{R}} \right\rceil, & \text{otherwise.} \end{cases}$$

where idx is the address of the last written page and \mathfrak{R} a constant indicating the maximum number of data records per page. It then deploys the *LBSearch*(t_s, idx_{lb}) procedure as illustrated in Algorithm 2. It is easy to see that in each recursion step, *LBSearch* always moves clockwise (increasing time order) and that $idx_{lb} \leq idx_{t_q}$.

Algorithm 2 *LBSearch* (No Anchors)

Input: t_q : the query (timestamp), *current*: begin search address

Output: The page that contains t_q .

```

1: procedure LBSEARCH( $t_q, current$ )
2:    $p = readPage(current)$ ;
3:   if (isIndexPage( $p$ )) then
4:     // logical right shift
5:     return LBSearch( $t_q, current + 1$ );
6:   else
7:      $t_1 = P.record[0].ts$ ;
8:      $t_2 = P.record[P.lbu].ts$ ;
9:     if ( $t_1 \leq t_q \leq t_2$ ) then
10:      return  $P$ ;
11:   end if
12:   return LBSearch( $t_q, current + idx_{lb}(t_q, t_2)$ );
13: end if
14: end procedure

```

It is important to note that a lower bound can only be estimated if the fetched page, on each step of the recursion, contains a timestamp value. Our discussion so

far, assumes that the only pages that carry a timestamp are data pages which contain a sequence of data records $\{[ts_1, val_1] \dots [ts_{\mathfrak{R}}, val_{\mathfrak{R}}]\}$. In such a case, the *LBSearch* has to *shift* right until a data page is located. In our experiments we noted that this deficiency could add in some cases 3-4 additional page reads. In order to correct the problem we store the last known timestamp inside each index page (named *Anchor*).

ii) *ScaleSearch*: When index pages are uniformly spread out across the flash media, then a more aggressive search strategy might be more effective. In *ScaleSearch*, which is the technique we deployed in *MicroHash*, instead of using idx_{lb} in the first step we use idx_{scaled} :

$$idx_{scaled}(t_q, t_s) = \begin{cases} \left\lceil \frac{t_q - t_a}{t_b - t_a} * idx \right\rceil, & \text{if cycle} == 0; \\ idx + \left\lceil \frac{t_q - t_a}{t_b - t_a} * n \right\rceil, & \text{otherwise.} \end{cases}$$

We then use *LBSearch* in order to refine the search. Note that idx_{scaled} might in fact be larger than idx_{t_q} in which case *LBSearch* might need to move counter-clockwise (decreasing time order).

Performing a timestamp-based range query $Q(t_q, a, b)$ is a simple extension of the equality search. More specifically, we first perform a *ScaleSearch* for the upper bound b (i.e. $Q(t_q, b)$) and then sequentially read backwards until a is found. Note that data pages are chained in reverse chronological order (i.e. each data page maintains the address of the previous data page) and therefore this operation is very simple.

6.3 Search Optimizations

In the basic *MicroHash* approach, index pages on flash might not be fully occupied. This incurs a significant performance penalty when somebody performs a search by value, because the system has to read in memory more pages than necessary. In this section we present two alternative methods that alleviate this performance penalty. The first method, named *Elf-Like Chaining* (*ELC*), eliminates non-full index pages which as a result decreases the number of pages required to answer a query, while the second method, named *Two-Phase Read* minimizes the number of bytes transferred from the flash media.

6.3.1 Elf-Like Chaining (ELC)

In the *MicroHash* index, pages are chained using a back-pointer as illustrated in Figure 5 (named *MicroHash Chaining*). Inspired from the update policy of the ELF filesystem [2], we also investigate, and later experimentally evaluate, the *Elf-like Chaining* (*ELC*) mechanism. The objective of *ELC* is to create a linked list in which

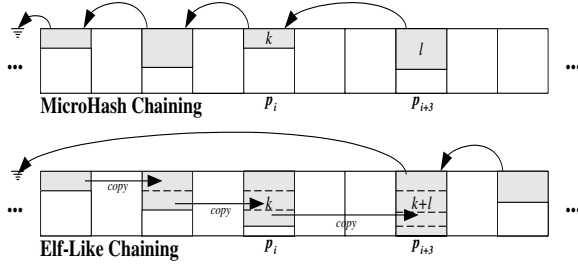


Figure 5: Index Chaining Methods: a) MicroHash Chaining and b) ELF-like Chaining.

each node, other than the last node, is completely full. This is achieved by copying the last non-full index page into a newer page, when new index records are requested to be added to the index. This procedure continues until an index page becomes full, at which point it is not further updated.

To better understand the two techniques, consider the following scenario (see Figure 5): An index page on flash (denoted as p_i ($i \leq n$)), contains k ($k < p_i^{size}$) index records $\{ir_1, ir_2, \dots, ir_k\}$ that in our scenario map to directory bucket v . Suppose that we create a new data page on flash at position p_{i+1} . This triggers the creation of l additional index records, which in our scenario map to the same bucket v . In **MicroHash Chaining (MHC)**, the buffer manager simply allocates a new index page for v and keeps the sequence $\{ir_1, ir_2, \dots, ir_l\}$ in memory until the LRU replacement policy forces the page to be written out. Assuming that the new index sequence is forced out of memory at p_{i+3} , then p_i will be back-pointed by p_{i+3} as shown in Figure 5. In **Elf-Like Chaining (ELC)**, the buffer manager reads p_i in memory and then augments it with the l new index records (i.e. $\{ir_1, \dots, ir_k, \dots, ir_{k+l}\}$). However, p_i is not updated due to the write and wear constraint, but instead the buffer manager writes the new $l + k$ sequence to the end of the flash media (i.e. at p_{i+3}). Note that p_i is now not backpointed by any other page and will not be utilized until the block delete, guided by the *idx* pointer, erases it.

The optimal compaction degree of index pages in *ELC* significantly improves the search performance of an index as it is not required to iterate over partially full index pages. However, in the worse case, *ELC* might introduce an additional page read per indexed data record. Additionally we observed in our experiments, presented in Section 8, that *ELC* requires on average 15% more space than the typical MicroHash chaining. In the worst case, the space requirement of *ELC* might double the requirement of *MHC*.

Consider again the scenario under discussion. This time assume that the buffer manager reads p_i in memory

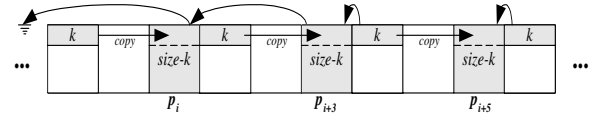


Figure 6: Sequential Trashing in ELC.

and then augments p_i^{size} (a full page) new index records as shown in Figure 6. That will evict p_i to some new address (in our scenario p_{i+3}). However some additional $p_i^{size} - k$ records are still in the buffer. Assume that these pages are at some point evicted from memory to some new flash position (in our scenario p_{i+5}). So far we utilized three pages (p_i , p_{i+3} and p_{i+5}) while the index records could fit into only 2 index pages (i.e. $k + p_i^{size}$ records, $k < p_i^{size}$). When the same scenario is repeated, then we say that *ELC* suffers from *Sequential Trashing* and *ELC* will require double the required space to accommodate all index records.

6.3.2 Two-Phase Page Reads

Our discussion so far assumes that pages are read from the flash media on a page-by-page basis (usually 512B per page). When pages are not fully occupied, such as index pages, then a lot of empty bytes (padding) is transferred from the flash media to memory. In order to alleviate this burden, in [1] we exploit the fact that reading from flash can be performed at any granularity (i.e. as small as a single byte). Specifically, we propose the deployment of a *Two-Phase Page Read* in which the MCU reads a fixed header from flash in the first phase, and then reads the exact amount of bytes in the next phase. We experimentally evaluated the performance of two-phase reads versus single phase reads using the RISE sensor node and found that such an approach significantly minimizes energy consumption.

7 Experimental Methodology

In this section we describe the details of our experimental methodology.

7.1 Experimental Testbed

We have implemented MicroHash along with a tiny LRU BufferManager in nesC[6], the programming language of TinyOS[7]. TinyOS is an open-source operating system designed for wireless embedded sensor nodes. It was initially developed at UC-Berkeley and has been deployed successfully on a wide range of sensors including the RISE mote. TinyOS uses a component-based architecture that enables programmers to wire together in on-

demand basis the minimum required components. This minimizes the final code size and energy consumption as sensor nodes are extremely power and memory limited. nesC [6] is the programming language of TinyOS and it realizes its structuring concepts and its execution model.

Our implementation consists of approximately 5000 lines of code and requires at least 3KB in SRAM. Specifically we use one page as a write buffer, two pages for reading (i.e. one for an index page and one for a data page), one page as an indexing buffer, one for the directory and one final page for the root page. In order to increase insertion performance and index page compactness, we also supplement additional index buffers (i.e. 2.5KB-5KB).

We had to write a library that simulates the flash media using an operating system file, in order to run our code in TOSSIM [8], the simulation environment of TinyOS. We additionally wrote a library that intercepts all messages communicated from TinyOS to the flash library and prints out various statistics and one final library that visualizes the flash media using bitmap representations.

7.2 PowerTOSSIM - Energy Modeling

PowerTOSSIM is a power modeling extension to TOSSIM presented in [14]. In order to simulate the energy behavior of the RISE sensor we extended PowerTOSSIM and added annotations to the MicroHash structure that accurately provide information when the power states change in our environment. We have focused our attention on precisely capturing the flash performance characteristics as opposed to capturing the precise performance of other less frequently used modules (the radio stack, on-chip flash, etc).

Our power model follows our detailed measurements of the RISE platform [1], which are summarized as following: We use a 14.8 MHz 8051 core operating at 3.3V with the following current consumption 14.8mA (On), 8.2mA (Idle), 0.2 μ A (Off). We utilize a 128MB flash media with a page size of 512B and a block size of 16KB. The current to read, write and block delete was 1.17mA, 37mA and 57 μ A and the time to read in the three pre-mentioned states was 6.25ms, 6.25ms, 2.27ms.

Using these parameters, we performed an extensive empirical evaluation of our power model and found that PowerTOSSIM is indeed a very useful and quite accurate tool for modeling energy in a simulation environment. For example we measured the energy required to store 1 MB of raw data on an RISE mote and found that this operation requires 1526mJ while the same operation in our simulation environment returned 1459mJ, which has a error of only 5%. On average we found that PowerTOSSIM provided an accuracy of 92%.

7.3 Dataset Descriptions

Since we cannot measure environmental conditions, such as temperature or humidity in a simulation environment, we adopt a trace-driven experimental methodology in which a real dataset is fed into the TOSSIM simulator. More specifically, we use the following datasets:

Washington State Climate: This is a real dataset of atmospheric data collected by the Department of Atmospheric Sciences at the University of Washington [21]. Our 268MB dataset contains readings on a minute basis between January 2000 and February 2005. The readings, which are recorded at a weather logging station in Washington, include barometric pressure, wind speed, relative humidity, cumulative rain and others. Since many of these readings are not typically measured by sensor nodes we only index the temperature and pressure readings, and use the rest readings as part of the data included in a record. Note that this is a realistic assumption, as sensor nodes may concurrently measure a number of different parameters.

Great Duck Island (GDI 2002): This is a real dataset from the habitat monitoring project on the Great Duck Island in Maine [15]. We use readings from one of the 32 nodes that were used in the spring 2002 deployment, which included the following readings: light, temperature, thermopile, thermistor, humidity and voltage. Our dataset includes approximately 97,000 readings that were recorded between October and November 2002.

8 Experimental Evaluation

In this section we present extensive experiments to demonstrate the performance effectiveness of the MicroHash Index structure. The experimental evaluation described in this section focuses on three parameters: i) **Space Overhead**, of maintaining the additional index pages, ii) **Search Performance**, which is defined as the average number of pages accessed for finding the required record and iii) **Energy Consumption**, for indexing the data records. Due to the design of the MicroHash index, each page was written exactly once during a cycle. Therefore there was no need to experimentally evaluate the wear-leveling performance.

8.1 Overhead of Index Pages

In the first series of experiments we investigate the overhead of maintaining the additional index pages on the flash media. For this reason we define the overhead ratio Φ as follows: $\Phi = \frac{Index\ Pages}{Data\ Pages + Index\ Pages}$. We investigate the parameter Φ using a) An increasing buffer size and b) An increasing data record size.

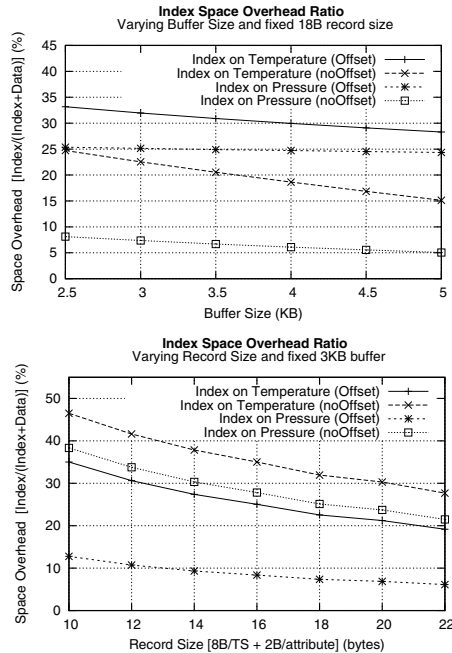


Figure 7: Space Overhead of Index Pages with a) varying buffer size and b) varying record size.

We also evaluate two different index record layouts: a) *Offset*, in which an index record has the following form {pageid,offset} and *NoOffset*, in which an index record has the form {pageid}. We use the five year timeseries from the Washington state climate dataset and index data records based on their temperature and pressure attribute. The data record on each of the 2.9M time instances was 18 bytes (i.e. 8B timestamp + 5x2B readings).

8.1.1 Increasing Buffer Size

Figure 7 (top) presents our results using a varying buffer. The figure shows that in all cases a larger buffer helps in fitting more index records per page which therefore also linearly reduces the overall space overhead. In both the pressure and temperature case, the *noOffset* index record layout significantly reduces the space overhead as less information is required to be stored inside an index record.

The figure shows that indexing on pressure achieves a lower overhead. This is attributed to the fact that the pressure changes slower than the temperature over time. This leads to fewer evictions of index pages during the indexing phase which consequently also increases the index page occupancy. We found that a 3KB buffer suffices to achieve occupancy of 75-80% in index pages.

8.1.2 Increasing Data Record Size

Sensor nodes usually deploy a wide array of sensors, such as a photo sensor, magnetometer, accelerometer and others. Therefore the data record size on each time instance might be larger than the minimum 10B size (8B timestamp and 2B data value). Figure 7 (bottom) presents our results using a varying data record size. The figure shows that in all cases a larger data record size decreases the space overhead proportion. Therefore it does not become more expensive to store the larger data records on flash.

8.2 Searching By Timestamp

In the next experimental series, we investigate the average number of pages that must be read in order to find a record by its timestamp. We search for 100 timestamps selected uniformly at random, from the available five year range, and then calculate the average number of pages read. Note that if we did not use an index, and thus had only data records on the flash, then we could find the expected record in $O(1)$ time as we could manipulate the position of the record directly. However, this would also violate our wear leveling mechanism.

We evaluate the proposed search by timestamp methods LBSearch and ScaleSearch under two different index page layouts: a) *Anchor*, in which every index page stores the last known data record timestamp and b) *NoAnchor*, in which an index page does not contain any timestamp information.

Figure 8 shows our results using the Washington state climate dataset for both an index on Temperature (Figure 8 top) and an Index on Pressure (Figure 8 bottom). The figures show that using an anchor inside an index pages is a good choice as it usually reduces the number of page reads by two, while it does not present a significant space overhead (only 8 additional bytes). The figures also show that ScaleSearch is superior to LBSearch as it exploits the uniform distribution of index pages on the flash media. This allows ScaleSearch to get closer to the result, in the first step of the algorithm.

The figures finally show that even though the time window of the query is quite large (i.e. 5 years or 128MB), ScaleSearch is able to find a record by its timestamp in approximately 3.5-5 page reads. Given that a page read takes 6.25ms, this operation requires according to the RISE model only 22-32ms or 84-120μJ.

8.3 Searching by Value: MicroHash Chaining vs. ELF-Like Chaining

The cost of searching a particular value on the flash media is linear with respect to the size of the flash me-

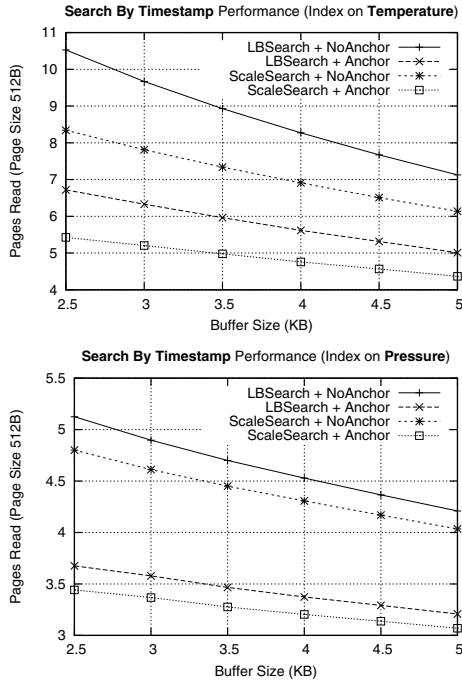


Figure 8: Search-By-Timestamp Performance of the MicroHash Index.

dia. However, a simple linear scan over 256 thousands pages found on a 128MB flash media, would result in an overwhelming large search cost. One factor that significantly affects search performance is the occupancy of index pages. In the basic MicroHash approach, index pages on the flash might not be fully occupied. If index pages are not fully utilized, then a search would require iterating over more pages than necessary.

In this section we perform an experimental comparison of the index chaining strategies presented in Section 6.3. We evaluate both MicroHash Chaining (MHC) and Elf-like chaining (ELC) using a fixed 3KB buffer. We deploy the chaining methods when the temperature is utilized as the index (we obtained similar results for pressure). Our evaluation parameters are : a) Indexing Performance (pages written) and b) Search Performance (pages read).

Figure 9 (top) shows that MHC always requires less page writes than ELC. The reason is that ELC's strategy results in about 15% sequential trashing, which is the characteristic presented in Section 6.3. Additionally, ELC requires a large number of page reads in order to replicate some of the index records. This is presented in the ELC Total plot, which essentially shows that it requires as many page reads as page writes in order to index all records. On the other hand, ELC's strategy results in linked lists of fully occupied index pages than MHC. This has as a result, an improved search perfor-

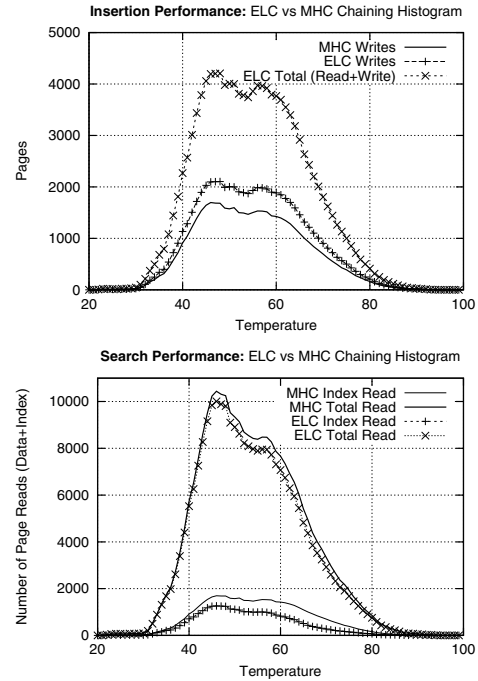


Figure 9: Comparing MicroHash Chaining (MHC) with ELF-like Chaining (ELC) using a) Insertion Performance and b) Searching Performance by Value.

mance since the system is required to fetch less index pages during search. This can be observed in Figure 9 (bottom), in which we present the number of index pages read and the total number of pages (index + data). On the other hand, we also observe that ELC only reduces the overall read gain to about 10%. This happens because the reading of data page, dominates the overall reading cost. However when searches are more frequent, then the 10% is still an advantage and therefore ELC is more appropriate than its counterpart MHC.

8.4 Great Duck Island Trace

In this last experimental series we index measurements from the great duck island study, described in Section 7.3. For this study we allocate a fixed 3KB index buffer along with a 4MB flash media that has adequate space to store all the 97,000 20-byte data readings.

In each run, we index on a specific attribute (i.e. Light, Temperature, Thermopile, Thermistor, Humidity and Voltage). We then record the overhead ratio of index pages Φ , the energy required by the flash media to construct the index as well as the average number of page reads that were required to find a record by its timestamp. We omit the search by value results, due to lack of space, but the results are very similar to those presented in the previous subsection.

Index On Attribute	Overhead Ratio Φ %	Energy Index (mJ)	ScaleSearch Page Reads
Light	26.47	4,134	4.45
Temperature	27.14	4,172	5.45
Thermopile	24.08	4,005	6.29
Thermistor	14.43	3,554	5.10
Humidity	7.604	3,292	2.97
Voltage	20.27	3,771	4.21

Table 2: Performance Results from Indexing and Searching the Great Duck Island dataset.

Table 2 shows that the index pages never require more than 30% more space on the flash media. For some readings that do not change frequently (e.g. humidity), we observe that the overhead is as low as 8%. The table also shows that indexing the records has only a small increase in energy demand. Specifically, the energy cost of storing the records on flash without an index was 3042mJ, which is on average only 779mJ less than using an index. Therefore maintaining the index records does not impose a large energy overhead. Finally the table shows that we were able to find any record by its timestamp with 4.75 page reads on average.

9 Related Work

There has been a lot of work in the area of query processing, in-network aggregation and data-centric storage in sensor networks. To the best of our knowledge, our work is the first that addresses the indexing problem on sensor nodes with flash memories.

A large number of flash-based file systems have been proposed in the last few years, including the Linux compatible Journaling Flash File System (JFFS and JFFS2)[17], the Virtual File Allocation Table (VFAT) for Windows compatible devices and the Yet Another Flash File System (YAFFS)[18], specifically designed for NAND flash with it being portable under Linux, uClinux, and Windows CE. The first file system for sensor nodes was Matchbox and this is provided as an integral part of the TinyOS [7] distribution. Recently the Efficient Log Structured Flash File System (ELF)[2] shows that it offers several advantages over Matchbox including higher read throughput and random access by timestamp. Other filesystems for embedded microcontrollers that utilize flash as a storage medium include the Transactional Flash File System (TFFS) [5]. However the main job of a file system is to organize the blocks of the storage media into files and directories and to provide transaction semantics on these attributes. Therefore a filesystem does not support the retrieval of records by their value as we do in our approach.

An R-tree and B-Tree index structure for flash memory on portable devices, such as PDA's and cell phones, has been proposed in [22] and [23] respectively. These structures use an in-memory address translation table, which hides the details of wear-leveling mechanism. However, such a structure has a very large footprint (3-4MB) which constitutes it inapplicable in the context of microcontrollers with limited SRAM.

Wear-Leveling techniques have also been reported by flash card vendors such as Sandisk [20]. These techniques are executed by a microcontroller which is located inside the flash card. The Wear-Leveling techniques are only executed within 4MB zones and are thus *local* rather than *global* which is the case in MicroHash. A main drawback of the *local* wear-leveling techniques is that the writes are no longer spread out uniformly across all available pages. Finally these techniques assume a dedicated controller while our techniques can be executed by the microcontroller of the sensor device.

Systems such as TinyDB[11] and Cougar[16] are optimized for sensor nodes with limited storage and relatively shorter-epochs, while our techniques are designated for sensors with larger external flash memories and longer epochs. Note that in TinyDB users are allowed to define fixed size materialization points through the STORAGE POINT clause. This allows each sensor to gather locally in a buffer some readings, which cannot be utilized until the materialization point is created in its entirety. Therefore even if there was enough storage to store MBs of data, the absence of efficient access methods makes the retrieval of the desired values expensive.

10 Conclusions

In this paper we propose the *MicroHash* index which is an efficient external memory hash index that addresses the distinct characteristics of flash memory. We also provide an extensive study of NAND flash memory when this is used as a storage media of a sensor device. Our design can serve several applications, including sensor and vehicular networks, which generate temporal data and utilize flash as the storage medium. We expect that our proposed access method will provide a powerful new framework to cope with new types of queries, such as temporal or top-k, that have not been addressed adequately to this date. Our experimental evaluation with real traces from environmental and habitat monitoring show that the structure we propose is both efficient and practical.

Acknowledgements

We would like to thank Joe Polastre (UC Berkeley) for providing us the Great Duck Island data trace and Victor

Shnayder (Harvard) for his help on the PowerTOSSIM environment. We would also like to thank Abhishek Mitra and Anirban Banerjee (UC Riverside) for their assistance in the RISE micro-benchmarks. Finally, we would like to thank our shepherd, Andrea Arpaci-Dusseau, and the anonymous reviewers for their numerous helpful comments. This work was supported by grants from NSF ITR #0220148, #0330481.

References

- [1] Banerjee A., Mitra A., Najjar W., Zeinalipour-Yazti D., Kalogeraki V. and Gunopulos D., "RISE Co-S : High Performance Sensor Storage and Co-Processing Architecture", In IEEE SECON, Santa Clara, CA, USA, to appear in 2005.
- [2] Dai H., Neufeld M., Han R., "ELF: an efficient log-structured flash file system for micro sensor nodes", In SenSys, Baltimore, pp. 176-187, 2004.
- [3] Dipert B., Levy M., "Designing with Flash Memory", AnnaBooks Publisher, 1994.
- [4] Fagin R., Nievergelt J., Pippenger N., Strong H.R.: "Extendible Hashing - A Fast Access Method for Dynamic Files", In ACM TODS, vol 4(3), pp. 315-344, 1979.
- [5] Gal E. and Toledo S., "A Transactional Flash File System for Microcontrollers", In USENIX 2005, Anaheim, CA, pp. 89-104, 2005.
- [6] Gay D., Levis P., Von Behren R. Welsh M., Brewer E. and Culler D., "The nesC Language: A Holistic Approach to Networked Embedded Systems", In ACM PLDI, San Diego, pp. 1-11, 2003.
- [7] Hill J., Szewczyk R., Woo A., Hollar S., Culler D., Pister K., "System Architecture Directions for Networked Sensors", In ASPLOS, Cambridge, MA, pp. 93-104, 2000.
- [8] Levis P., Lee N., Welsh M., and Culler D., "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications", In ACM SenSys, Los Angeles, CA, 2003.
- [9] Litwin W., "Linear Hashing: A New Tool for File and Table Addressing", VLDB 1980: 212-223.
- [10] Lymberopoulos D., Savvides A., "XYZ: A Motion-Enabled, Power Aware Sensor Node Platform for Distributed Sensor Network Applications", In IPSN, Los Angeles, pp. 449-454, 2005.
- [11] Madden S.R., Franklin M.J., Hellerstein J.M., Hong W., "The Design of an Acquisitional Query Processor for Sensor Networks", In ACM SIGMOD, San Diego, CA, USA, pp. 491-502, 2003.
- [12] Madden S.R., Franklin M.J., Hellerstein J.M., Hong W., "TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks", In OSDI, Boston, MA, pp. 131-146, 2002.
- [13] Sadler C., Zhang P., Martonosi M., Lyon S., "Hardware Design Experiences in ZebraNet", In ACM SenSys, Baltimore, pp. 227-238, 2004.
- [14] Shnayder V., Hempstead M., Chen B., Werner-Allen G., and Welsh M., "Simulating the Power Consumption of Large-Scale Sensor Network Applications", In ACM SenSys, pp. 188-200, 2004.
- [15] Szewczyk R., Mainwaring A., Polastre J., Anderson J., Culler D., "An Analysis of a Large Scale Habitat Monitoring Application", In ACM SenSys, Baltimore, MD, pp. 214-226, 2004.
- [16] Yao Y., Gehrke J.E., "Query Processing in Sensor Networks", In CIDR, Asilomar, CA, USA, 2003.
- [17] Woodhouse D. "JFFS : The Journalling Flash File System" Red Hat Inc., Available at: <http://sources.redhat.com/jffs2/jffs2.pdf>
- [18] Wookey "YAFFS - A filesystem designed for NAND flash", Linux 2004 Leeds, U.K.
- [19] Ramakrishnan R., Gehrke J., Database Management Systems, McGraw-Hill, Third edition, 2002.
- [20] "Sandisk Flash Memory Cards - Wear Leveling", October 2003 White Paper, Available at: <http://sandisk.com/pdf/oem/WPaper-WearLevelv1.0.pdf>
- [21] Live From Earth and Mars Project University of Washington, Seattle <http://www-k12.atmos.washington.edu/k12/grayskies/>
- [22] Wu C-H., Chang L-P., Kuo T-W., "An Efficient R-tree Implementation Over Flash-Memory Storage Systems", In RTCSA, Taiwan, pp. 409-430, 2003.
- [23] Wu C-H., Chang L-P., Kuo T-W., "An Efficient B-Tree Layer for Flash Memory Storage Systems", In RTCSA, New Orleans, pp. 17-24, 2003.
- [24] Xu N., Rangwala S., Chintalapudi K., Ganesan D., Broad A., Govindan R. and Estrin D., "A Wireless Sensor Network for Structural Monitoring", In Sensys, Baltimore, MD, pp. 13-24, 2004.

Adaptive Data Placement for Wide-Area Sensing Services

Suman Nath*
Microsoft Research
sumann@microsoft.com

Phillip B. Gibbons
Intel Research Pittsburgh
phillip.b.gibbons@intel.com

Srinivasan Seshan
Carnegie Mellon University
srini@cs.cmu.edu

Abstract

Wide-area sensing services enable users to query data collected from multitudes of widely distributed sensors. In this paper, we consider the novel distributed database workload characteristics of these services, and present IDP, an online, adaptive data placement and replication system tailored to this workload. Given a hierarchical database, IDP automatically partitions it among a set of networked hosts, and replicates portions of it. IDP makes decisions based on measurements of access locality within the database, read and write load for individual objects within the database, proximity between queriers and potential replicas, and total load on hosts participating in the database. Our evaluation of IDP under real and synthetic workloads, including flash crowds of queriers, demonstrates that in comparison with previously-studied replica placement techniques, IDP reduces average response times for user queries by up to a factor of 3 and reduces network traffic for queries, updates, and data movements by up to an order of magnitude.

1 Introduction

Emerging *wide-area sensing services* [18,26,27] promise to instrument our world in great detail and produce vast amounts of data. For example, scientists already use such services to make observations of natural phenomena over large geographic regions [2,5]; retailers, such as Walmart [9], plan to monitor their inventory using RFID tags; and network operators (ISPs) monitor their traffic using a number of software sensors [16]. A key challenge that these services face is managing their data and making it easily queriable by users. An effective means for addressing this challenge is to store the vast quantity of data in a wide-area distributed database, which efficiently handles both updates from geographically dispersed sensors and queries from users anywhere in the world [14].

Like traditional distributed databases, sensing service databases must carefully replicate and place data in order to ensure efficient operation. Replication is necessary for avoiding hot spots and failures within the system, while careful data placement is required for minimizing network traffic and data access latency. Although replication and data placement have been extensively studied in the

context of many wide-area systems [10, 12, 24, 32, 36, 37, 40, 41], existing designs are ill-suited to the unique workload properties of sensing services. For example, unlike traditional distributed databases, a sensing service database typically has a *hierarchical organization* and a *write-dominated* workload. Moreover, the workload is expected to be *highly dynamic* and to exhibit *significant spatial read and/or write locality*. These unique workload properties, discussed in further detail in this paper, significantly complicate the design of replication and placement techniques. For example, although replicating data widely can reduce query response times by spreading client load among more host machines and providing replicas closer to queriers, it also increases network traffic for creating replicas and propagating the large volume of writes to them. Similarly, placing data near the sources of reads/writes can reduce network overheads, but it also increases latency and CPU load for answering the hierarchical queries common in sensing services.

In this paper, we present an online, adaptive data placement and replication scheme tailored to the unique workload of wide-area sensing services. This scheme, called *IrisNet Data Placement (IDP)*, has been developed in the context of the IrisNet wide-area sensing system [8]. With IDP, each host efficiently and locally decides when to off-load data to other hosts and what data to off-load. The target hosts are selected such that data accessed together tend to reside on the same host. IDP makes such online decisions based on workloads, locality of queries, proximity of queriers and candidate replica locations, and total load on the hosts running the service. In making these decisions, IDP tries to balance three performance metrics: response time, bandwidth used for creating replicas, and bandwidth used to keep replicas up-to-date, in a way particularly suited to wide-area sensing service workloads. Such automatic and adaptive techniques are essential for a large-scale, widely-distributed sensing service because service administrators cannot be expected to determine and maintain an effective partitioning across all the hosts.

IDP explicitly detects flash crowds (*i.e.*, sudden bursts in queries to spatially localized data), replicates in response to them, and sheds load to newly created replicas, but only as long as the flash crowd persists. Thereafter, if load on the new replicas drops, unneeded replicas are destroyed. IDP places new replicas in proximity to the sources of queries and updates, but routes queries using a

*Work done while the first author was a PhD student at CMU and an intern at Intel Research Pittsburgh.

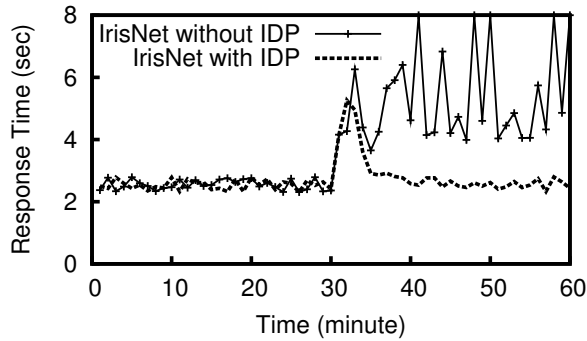


Figure 1: Result of injecting a real query workload into a 310 PlanetLab host deployment of IrisNet with and without IDP. A flash crowd is emulated at time = 30 minutes by increasing query rate by 300 times. A response time of 8 second represents a timeout.

combination of proximity routing and randomization, to ensure that queries favor nearby replicas but new replicas can succeed in drawing query load away from preexisting closer replicas.

We implemented IDP within IrisNet and evaluated its performance under real and synthetic sensing system workloads. Figure 1 presents a sample result demonstrating IDP’s effectiveness (details of the experimental setup will be described in Section 7). As shown, IDP helps IrisNet tolerate a flash crowd gracefully, keeping the response times close to their pre-flash crowd levels. Without IDP, IrisNet quickly becomes overloaded, resulting in higher response times and frequent query timeouts. Further results show the effectiveness of various components of IDP and provide a comparison with two previously-studied adaptive placement schemes [36, 41]. Compared to these schemes, IDP reduces average response time for user queries by up to a factor of 3 and reduces network traffic by up to an order of magnitude, while serving user queries, propagating new sensor data to replicas, and adjusting data placements.

In the remainder of the paper, we describe the characteristics of wide-area sensing service workloads and the architecture of a wide-area sensing system, present the algorithms that comprise IDP, describe our implementation of IDP, evaluate IDP through real deployment and simulation, briefly discuss related work, and conclude by summarizing our findings.

2 Wide-Area Sensing Systems

Although IDP has been developed in the context of IrisNet, the techniques are tailored more to the workload than to the specific sensing system, and hence should be effective for many wide-area sensing systems. In this section, we describe the workload in further detail by considering a few example wide-area sensing services. Then we

present an architecture for a generic sensing system on which these services can be built; IDP will be discussed in the context of this generic system, in order to emphasize its applicability beyond IrisNet.

2.1 Wide-Area Sensing Services

We consider representative services from five classes of wide-area sensing services, as summarized in Table 1:

Asset Tracker: This service is an example of a class of services that keeps track of objects, animals or people. It can help locate lost items (pets, umbrellas, *etc.*), perform inventory accounting, or provide alerts when items or people deviate from designated routes (children, soldiers, *etc.*). Tracking can be done visually with cameras or by sensing RFID tags on the items. For large organizations such as Walmart or the U.S. Military, the global database may be TBs of data.

Ocean Monitor: This service is an example of a class of environmental monitoring services that collect and archive data for scientific studies. A concrete example is the Argus coastal monitoring service [2, 18], which uses a network of cameras overlooking coastlines, and analyzes near-shore phenomena (riptides, sandbar formation, *etc.*) from collections of snapshot and time-lapse images.

Parking Finder: This service monitors the availability of parking spaces and directs users to available spaces near their destination (see, *e.g.*, [14, 18]). It is an example of a class of services that monitors availability or waiting times for restaurants, post offices, highways, *etc.* Queriers tend to be within a few miles of the source of the sensed data, and only a small amount of data is kept for each item being monitored.

Network Monitor: This service represents a class of distributed network monitoring services where the “sensors” monitor network packets. A key feature of this service is the large volume of writes. For example, a network monitoring service using NetFlow [4] summarization at each router can generate 100s of GBs per day [16]. We model our Network Monitor service after IrisLog [7, 18]. IrisLog maintains a multi-resolution history of past measurements and pushes the distributed queries to the data.

Epidemic Alert: This service monitors health-related phenomena (number of people sneezing, with fevers, *etc.*) in order to provide early warnings of possible epidemics (see, *e.g.*, [3]). When the number of such occurrences significantly exceeds the norm, an alert is raised of a possible epidemic. We use this service to consider a *trigger-based* read-write workload in which each object in the hierarchy reports an updated total to its parent whenever that total has changed significantly, resulting in a possible cascading of updates. Trigger-based workloads have not been considered previously in the context of data placement, but they are an important workload for sensing services.

	Asset Tracker	Ocean Monitor	Parking Finder	Network Monitor	Epidemic Alert
<i>Global DB Size</i>	large	very large	medium	large	medium
<i>Write Rate</i>	medium	low	low	very high	very low
<i>Read/Write Ratio</i>	low	very low	low	very low	one
<i>Read Burstiness</i>	low	very low	high	low	very low
<i>Read Skewness</i>	building, skewed	root, uniform	neighborhd, skewed	domain, uniform	uniform, uniform
<i>R/W Proximity</i>	neighborhood	root	neighborhood	domain	root
<i>Consistency</i>	variable: secs–mins	relaxed: \approx mins	variable: secs–mins	strict: \approx secs	relaxed: \approx mins

Table 1: Five representative wide-area sensing services and their characteristics.

A Qualitative Characterization. In Table 1, we characterize qualitatively the five representative services along seven dimensions. *Global DB Size* is the size of the global database for the service. Note that for Ocean Monitor, it includes space for a full-fidelity historical archive, as this is standard practice for oceanography studies. For Network Monitor, it includes space for a multi-resolution history. In order to provide different points of comparison, for the remaining services, we assume only the most recent data is stored in the database.¹ *Write Rate* is the rate at which objects are updated in the global database. We assume services are optimized to perform writes only when changes occur (e.g., Parking Finder only writes at the rate parking space availability changes). *Read/Write Ratio* is the ratio of objects read to objects written. A distinguishing characteristic of sensing services is that there are typically *far more writes than reads*, as far more data is generated than gets queried. An exception in our suite is Epidemic Alert, because we view each triggered update as both a read and a write. *Read Burstiness* refers to the frequency and severity of flash crowds, i.e., large numbers of readers at an object in a short window of time. For example, Parking Finder has flash crowds right before popular sporting events. *Read Skewness* is measured in two dimensions: the geographic scope of the typical query (e.g., parking queries are often at the neighborhood level) and the uniformity across the objects at a given level (e.g., parking queries are skewed toward popular destinations). *R/W Proximity* reflects the typical physical distance from the querier to the sensors being queried, as measured by the lowest level in the hierarchy they have in common. For example, an oceanographer may be interested in coastlines halfway around the world (root level proximity), but a driver is typically interested only in parking spaces within a short drive (neighborhood level proximity). *Consistency* refers to the typical query’s tolerance for stale (cached) data.

2.2 Sensing System Architecture

We now describe the architecture of a generic wide-area sensing system on which the above services can be built. At a high level, a sensing system consists of three main

components.

Data Collection. This component collects data from the sensors, which may be cameras, RFID readers, microphones, accelerometers, etc. It also performs *service-specific filtering* close to the sensors, in order to reduce high-bit-rate sensed data streams (e.g., video) down to a smaller amount of post-processed data. In IrisNet, for example, this component is implemented as *Sensing Agents* (SAs) that run on Internet-connected hosts, each with one or more attached sensors. The post-processed data is stored as *objects* in the data storage component, described next.

Data Storage. This component stores and organizes the data objects so that users can easily query them. Because of the large volume of data and because objects are queried far less frequently than they are generated, a scalable architecture must distribute the data storage so that objects are stored near their sources. Sensed data are typically accessed in conjunction with their origin geographic location or time. Organizing them hierarchically, e.g., according to geographic and/or political boundaries, is therefore natural for a broad range of sensing services. We assume that each sensing service organizes its data in its own hierarchies, tailored to its common access patterns. For example, the data of the Parking Finder may be organized according to a geographic hierarchy (for efficient execution of common queries seeking empty parking spots near a given destination), while data in an Ocean Monitor may further be organized according to timestamps (supporting queries seeking all riptide events in the year 2004).² Thus, the storage component of a sensing system provides a hierarchical distributed database for a sensing service. Let the *global database* of a service denote a sensor database built from the complete hierarchy for a service and the data collected from all the sensors used by that service. A host in the sensing system contains a part of this database, called its *local database*. Thus, each sensing service has exactly one global database, which can be distributed among multiple hosts. However, each host may own multiple local databases, one for each sensing service using the host.

In IrisNet, for example, this storage component is implemented as *Organizing Agents* (OAs) that run on

¹This assumption is not crucial for the results that follow.

²A service can have multiple hierarchies over the same data.

Internet-connected hosts. The global database is represented as a single (large) XML document whose schema defines the hierarchy for the service. Note that the hierarchy is logical—any subset of the objects in the hierarchy can be placed in a host’s local database.

Query Processing. A typical query selects a subtree from the hierarchy. The query is then routed over the hierarchy, in a top-down fashion, with the query predicates being evaluated at each intermediate host. The hierarchy also provides opportunities for in-network aggregation—as the data objects selected by the query are returned back through the hierarchy, each intermediate host aggregates the data sent by its children and forwards only the aggregated results. For example, a query for the total value of company assets currently within 100 miles of a given disaster is efficiently processed in-network by first summing up the total asset value within each affected building in parallel, then having these totals sent to each city object for accumulation in parallel, and so on up the object hierarchy. This technique, widely used in wireless sensor networks [31], is also used in a few existing wide-area sensing systems [14, 17, 39].

For example, in IrisNet, a query is specified in XPATH [43], a standard language for querying an XML document. Figure 2 shows a portion of an example hierarchy (for a Parking Finder service), a partitioning of the hierarchy among hosts (OAs), and an example of how queries are routed on that particular partitioning. The hierarchy is geographic, rooted at NE (the northeastern United States), continuing downward to PA (state), Pittsburgh (city), Oakland and Shadyside (neighborhoods), and then individual city blocks within those neighborhoods. In the figure, a single physical host holds the NE, PA, and Pittsburgh objects (*i.e.*, the data related to the corresponding regions). The (XPath) query shown is for all available parking spaces in either block 1 or block 3 of Oakland. The query is routed first to the Oakland host, which forwards subqueries to the block 1 and block 3 hosts, which respond to the Oakland host, and finally the answer is returned to the query front-end. In general, a query may require accessing a large number of objects in the hierarchy in a specific top-down order, and depending on how the hierarchy is partitioned and placed, the query may visit many physical hosts.

3 The Adaptive Data Placement Problem

In this section, we describe the adaptive data placement problem, motivated by the characteristics of wide-area sensing services.

3.1 Data Placement Challenges

Based on the discussion so far, we summarize the distinguishing challenges and constraints for adaptive data

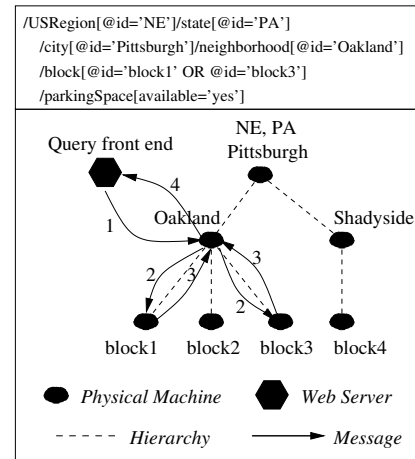


Figure 2: Top: An XPath query. Bottom: A mapping of the objects in the hierarchy to seven hosts, and the messages sent to answer the query (numbers depict their relative order).

placement for wide-area sensing services as follows:

Data Access Pattern. The data access patterns for wide-area sensing are far more complex than traditional distributed file systems or content distribution networks. First, the in-network aggregation within a hierarchically-organized database requires that different parts of the database be accessed in a particular order and the data be transmitted *between* the hosts containing the data. Moreover, in-network aggregation updates may be trigger-based. Second, different queries may select different, possibly overlapping, parts of the hierarchy. As a result, the typical granularity of access is unknown *a priori*.

Dynamic Workload. Since many different sensing services can share the same set of hosts, even the high-level access patterns to the hosts are not known *a priori*. Moreover, the read and write load patterns may change over time and, in particular, the read load may be quite bursty. For example, after an earthquake, there may be a sudden spike in queries over sensed data from the affected geographic region. The data placement should quickly adapt to workload dynamics.

Read/Write Workload. With millions of sensors constantly collecting new readings, the aggregate update rate of the sensor database is potentially quite high (*e.g.*, 100s of GB per day for Network Monitor). Moreover, the number of writes is often an *order of magnitude or more greater* than the number of reads. Thus, any placement must consider the locations of the writers—not just the readers—and the number of replicas is constrained by the high cost of updating the replicas.

Capacity Constraints. Each object has a weight (*e.g.*, storage requirement, read and write load) and each host has a capacity (*e.g.*, storage, processing power) that must be respected.

Wide-area Distribution. The sensors and queriers may

span a wide geographic area. On the other hand, many services have high R/W proximity. Thus, data placement must be done carefully to minimize communication costs and exploit any proximity.

Consistency and Fault Tolerance. Each service’s consistency and fault tolerance requirements must be met. Based on our representative services, we assume best effort consistency suffices. (Note that most data have only a single writer, *e.g.*, only one sensor updates the availability of a given parking space.) For fault tolerance, we ensure a minimum number of replicas of each object.

3.2 Problem Formulation and Hardness

To capture all these aspects, we formalize the data placement problem for wide-area sensing services as follows. *Given a set of global databases (one for each service), the hosts where data can be placed, a minimum number of copies required for each object, and the capacities (storage and computation) of the hosts, adapt to the dynamic workload by determining, in an online fashion, (1) the local databases, possibly overlapping, to be placed at different hosts, and (2) the assignments of these local databases to the hosts such that the capacity of each host is respected, the minimum number of copies (at least) is maintained for each object, the average query latency is low, and the wide-area traffic is low.*

Given the complexity of this task, it is not surprising that even dramatically simplified versions of this problem are NP-hard. For example, even when there is an unbounded number of hosts, all hosts have the same capacity C , all pairs of hosts have the same latency, the query workload is known, and there are no database writes, the problem of fragmenting a global database into a set of fragments of size at most C , such that the average query latency is below a given threshold, is NP-hard.³

Because the global hierarchical database can be abstracted as a tree, selecting fragments is a form of graph (tree) partitioning and assigning fragments is a form of graph (tree) embedding. Many approximation algorithms have been proposed for graph partitioning and graph embedding problems (*e.g.*, in the VLSI circuit optimization literature [25, 38]). None of these proposed solutions address the complex problem we consider. However, we get the following two (intuitive) insights from the existing approximation algorithms, which we use in IDP. First, the final partitions are highly-connected clusters. If, as in our case, the given graph is a tree, each partition is a subtree of the graph. Second, if the edges are weighted (in our case, the weights reflect the frequency in which a hop in the object hierarchy is taken during query routing), and the objective is to minimize the cost of the edges between partitions, most of the heavy edges are within partitions.

³By a reduction from the Knapsack problem.

4 IDP Algorithms

Due to the complexity of optimal data placement, IDP relies on a number of practical heuristics that exploit the unique properties of sensing services. Our solution consists of three simplifications. First, each host selects what parts of its local database to off-load using efficient heuristics, instead of using expensive optimal algorithms. Second, each host decides when to off-load or replicate data independently, based on its local load statistics. Finally, to mitigate the suboptimal results of these local decisions, we use placement heuristics that aim to yield “good” global clustering of objects. These three components, called *Selection*, *Reaction*, and *Placement* respectively, are described below.

4.1 Fragment Selection

The *selection* component of IDP identifies the *fragments* of its local database (*i.e.*, the sets of objects) that should be transferred to a remote host. The fragments are selected so as to decrease the local load below a threshold while minimizing the wide-area traffic for data movement and queries. At a high level, fragment selection involves partitioning trees. However, previous tree partitioning algorithms [30, 38] tend to incur high computational costs with their $O(n^3)$ (n = number of objects) complexity, and hence prevent a host from shedding load promptly. For example, in IrisNet, using a 3 GHz machine with 1 GB RAM, the algorithms in [30, 38] take tens of minutes to optimally partition a hierarchical database with 1000 objects. Such excessive computational overhead would prevent IDP from shedding load in a prompt fashion. On the other hand, trivial algorithms (*e.g.*, the greedy algorithm in Section 7.1.1) do not yield “good” fragmentation. To address this limitation, we exploit properties of typical query workloads to devise heuristics that provide near optimal fragmentation with $O(n)$ complexity. We call our algorithm POST (*Partitioning into Optimal Sub-Trees*). As a comparison, under the same experimental setup used with the aforementioned optimal algorithms, POST computes the result in a few seconds. Below we describe the algorithm.

Database Partitioning with POST. We use the following terminology in our discussion. For a given host, let G_I denote the set of (*I*nternal) objects in the local database, and G_E denote the set of non-local (*E*xternal) objects (residing on other hosts) and the set of query sources. Define the *workload graph* (Figure 3) to be a directed acyclic graph where the nodes are the union of G_I and G_E , and the edges are pointers connecting parent objects to child objects in the hierarchy and sources to objects. Under a given workload, an edge in the workload graph has a weight corresponding to the rate of queries along that edge (the weight is determined by locally collected statis-

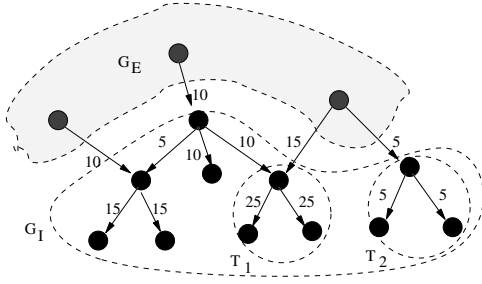


Figure 3: The workload graph of a host. G_I represents the local database and G_E is a set of objects on other hosts. The edges are labeled with the load on the corresponding edge. The circles labeled T_1 and T_2 represent two fragments of the local database of size 3.

tics, as described in Section 6.1). The weight of a node in G_I is defined as the sum of the weights of all its incoming edges (corresponding to its query load) and the weights of all its outgoing edges to nodes in G_E (corresponding to its message load).

For any set T of objects within G_I , we define T 's cost to be the sum of the weights of nodes in T . The $cut_{internal}$ of T is the total weight of the edges coming from some node in G_I to some node in T , and it corresponds to the additional communication overhead incurred if T were transferred. The $cut_{external}$ is the total weight of the edges coming from some node in G_E to some node in T , and it corresponds to the reduction of load on the node if T were transferred. In Figure 3, the $cut_{internal}$ of T_1 is 10, while the $cut_{external}$ is 15.

Intuitively, to select a good fragment, we should minimize $cut_{internal}$ (achieved by T_2 in Figure 3) so that it introduces the minimum number of additional subqueries or maximize $cut_{external}$ (achieved by T_1 in Figure 3) so that it is the most effective in reducing external load.

To design an efficient fragmentation algorithm for sensing services, we exploit the following important characteristics in their workloads: *A typical query in a hierarchical sensor database accesses all objects in a complete subtree of the tree represented by the database, and the query can be routed directly to the root of the subtree.* This observation is well supported by the IrisLog [7] query trace (more details in Section 7), which shows that more than 99% of user requests select a complete subtree from the global database. Moreover, users make queries on all the levels in the hierarchy, with some levels being more popular than the others. Under such access patterns, the optimal partition T is typically a subtree. The reason is that transferring only part of a subtree T from a host H_1 to another host H_2 may imply that a top-down query accesses objects in H_1 (the top of T) then in H_2 (the middle of T) and then back in H_1 (the bottom of T), resulting in a suboptimal solution.

The above observation enables POST to restrict the

search space and run in linear time. POST sequentially scans through all the nodes of the workload graph, and for each node it considers the whole subtree rooted at it. For all the subtrees with costs smaller than the given capacity C , it outputs the one with the optimal objective. The search space is further decreased by scanning the nodes in the workload-graph in a bottom-up fashion, thus considering the lower cost subtrees near the leaves before the higher cost subtrees further up the tree. As mentioned before, in typical settings, POST takes a few seconds to run. Yet, as we will show in Section 7.1.1, the quality of the resulting fragmentation, in practice, is very close to that of the $O(n^3)$ optimal algorithms that take tens of minutes to run.

IDP uses POST with different objective functions for different situations. For example, POST could choose the subtree that maximizes the value of $cut_{external}$. In Figure 3, T_1 denotes such a fragment. Replicating or splitting such fragments would be effective in reducing the external load on the host. Note that this objective tends to choose large fragments since $cut_{external}$ increases with size. Alternatively, POST could minimize the value of $cut_{internal}$. In Figure 3, T_2 denotes such a fragment. Splitting such fragments would minimize any resulting increase in the host hop counts of queries. Unfortunately, this objective tends to pick small fragments which may slow down the load shedding. To avoid this, POST only considers fragments of size greater than $C/2$, where C is a parameter (discussed next) specifying the cost of the fragments POST must select.

Parameters of POST. IDP must choose a value for C to pass to the POST algorithm. C can be chosen to be the smallest amount C_{min} of load whose removal makes the host underloaded. This minimizes the network traffic cost of data placement, but the host may become overloaded again easily. A C far greater than C_{min} would protect the host from overload but increase the overhead and duration of a load shedding event. This choice is equivalent to having two thresholds in the system: the load shedding is triggered when the load is above a *high-watermark* threshold Th_{high} , and at that point the load is shed until the load goes below a *low-watermark* threshold Th_{low} . In addition to these two thresholds, IDP uses a *deletion threshold* Th_{del} that characterizes the lowest load at an object replica that still justifies retaining that replica.

Load Estimation. Each host needs to monitor its load in order to decide when to start shedding load. The load could be estimated using an exponentially-decaying average, L_{avg} , of the instantaneous loads, L_i , on all the objects in the local database. However, after shedding load, L_{avg} overestimates the actual load for a while, because its time-decaying average is unduly inflated by the load just prior to shedding. As a result, unwarranted further shedding could occur repeatedly. A possible fix would be to prevent the host from shedding load until L_{avg} becomes

valid; however, this approach would react too slowly to increasing flash crowds. Instead, we adapt a technique used by TCP to estimate its retransmission time out values [20]. We maintain the exponentially-decaying average, V_{avg} , of the value $|L_{avg} - L_i|$, and then estimate the load as $L_{avg} - 2V_{avg}$, until L_{avg} again becomes valid. Our experiments show that this provides a stable estimate of a host's local load.

4.2 Reaction

In IDP, hosts exchange database fragments in order to adapt to changing workloads, using the following four operations:

- *Delete* removes a fragment from the local database. IDP ensures that each object maintains a minimum number of replicas (defined by the service author as a fault tolerance parameter), by not permitting deletes that would violate the specified minimum.
- *Coalesce* merges a fragment from a remote host into the local database. If the local database already contains some part of the fragment, it gets overwritten. *Proactive coalesce* retrieves a fragment and coalesces it. IDP uses this to reduce communication cost and thereby improve performance.
- *Split* deletes a fragment from the local database and sends it to a host where the fragment gets coalesced.
- *Replicate* creates a replica of a fragment and sends it to a host where the fragment gets coalesced.

Naturally, the appropriate action to take at a given time depends on the conditions at the hosts. The following summarizes five host conditions and the corresponding actions taken by IDP:

1. *Read/write load of an object is below Th_{del}* : IDP can delete the object if it has more than the minimum number of replicas.
2. *Read load of the host is above the threshold Th_{high}* : IDP must delete some objects (by applying the above action, as appropriate), replicate a fragment, or split the local database. As discussed in Section 4.1, IDP replicates or splits based on (1) whether POST maximizes $cut_{external}$ or $cut_{internal}$ and (2) the capacity of the target host to handle part or all of the fragment's load.
3. *A local database is too large*: IDP must delete some objects or split the database.
4. *Write load of the host is above Th_{high}* : IDP must replicate a fragment or split the database.
5. *The host is generating a lot of subqueries for an object owned by some other host*: To eliminate these subqueries, IDP proactively coalesces the object. If the host is overloaded as a result, IDP will take additional steps to shed load.

4.3 Fragment Placement

Given a fragment and its load information, IDP must find, through a suitable discovery mechanism, a host that can take the extra load. The simplest approach selects a random host capable of taking the extra load. However, this may require a query to visit multiple hosts, increasing query latency. Therefore, we use two heuristics that exploit the query and data source characteristics to improve the overall performance.

The first heuristic uses our previous observation that subtrees of the global database should be kept as clustered as possible. Thus the host discovery component first considers the *neighboring* hosts as possible destinations for a fragment. A host H_1 is a neighbor of fragment f if H_1 owns an object that is a parent or a child of some object in f . A host H_2 , trying to split or replicate a fragment, f , first sees if any of the neighboring hosts of f can take the extra load, and if such a host is found, f is sent to it. If more than one neighboring host is capable of taking the extra load, then the one having the highest *adjacency score* is chosen. The adjacency score of a host H_1 with respect to the fragment f is the expected reduction of read traffic (based on local statistics) if f is placed in H_1 . This helps maintain large clusters throughout the system.

When no suitable neighboring host is found, IDP's second heuristic searches for the host capable of taking the extra load that is the closest to the *optimal location* among all such hosts. The optimal location for an object is the weighted mid-point of its read and write sources. We account for the location of sources using the GNP [34] network mapping system. Specifically, each component of the sensing system (sensor, host, or client web interface) has GNP coordinates, and the cartesian distance between hosts is used to estimate the latency between them. If the read and write loads of an object are R and W , and the average read and write source locations are GNP_{read} and GNP_{write} , then the optimal location is given by $GNP_{opt} = \frac{R \cdot GNP_{read} + W \cdot GNP_{write}}{R + W}$. The optimal location of a fragment is the average of the optimal locations of all the objects in the fragment.

4.4 A Simple Run

Intuitively, IDP achieves its goal by letting hosts quickly readjust their loads by efficiently partitioning or replicating objects and by keeping the neighboring objects clustered. We illustrate a simple run of IDP in Figure 4. Rectangular boxes represent hosts and the trees represent their local databases. Initially the global database is placed at one host (1), which experiences high read and write load. To shed write load the host then determines two fragments (one containing object 5 and the other containing objects 3 and 6) for splitting. The fragments are then placed at newly-discovered hosts near the write sources (2). To shed read load, it then determines two more fragments

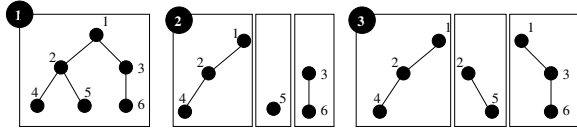


Figure 4: A simple adaptive data placement scenario.

and places them in hosts already owning objects adjacent to the fragments (3).

5 Replica Selection During Queries

While processing a query, if a host's local database does not contain some objects required by the query, subqueries are generated to gather the missing objects. Because the global database is split or replicated at the granularity of individual objects, IDP can route subqueries to *any* of the replicas of the required objects. Two simple replica selection strategies are *proximity selection* that selects the closest replica and *random selection* that selects a random replica. However, as we will show in Section 7, these simple strategies are not very effective for sensing services. For example, in the presence of read-locality, proximity selection fails to shed load from the closest (and the most overloaded) replica. Therefore, we use the following randomized selection algorithm.

Randomized Response-Time Selection. Another possible policy is to select the replica with the smallest expected response time (network latency plus the host processing time). Each host models the response time of the replicas using the moving averages of their previous actual response times. Note that when the load of all the replicas are similar (*i.e.*, their host processing times are comparable), the choice is determined by network latencies. This may cause load oscillation of the optimally placed host due to a herd effect [13]. To avoid this oscillation, we instead use the following randomized response-time (RRT) algorithm. Each host, when it contacts a replica r_i , measures its response times and uses them to estimate r_i 's expected response time e_i . Suppose a host is considering the replicas in increasing order of response time, r_1, r_2, \dots, r_n . The algorithm assigns a weight w_i to replica r_i according to where e_i falls within the range e_1 to e_n as follows: $w_i = (e_n - e_i) / (e_n - e_1)$. Finally it normalizes the weights to compute the selection probability p_i of r_i as $p_i = w_i / \sum_j w_j$. Note that if all the hosts are similarly loaded, RRT prefers nearby replicas. If the replicas are at similar distances, RRT prefers the most lightly-loaded host. In addition, the randomization prevents the herd effect.

6 Implementation Details

We now sketch our implementation of IDP in IrisNet, including the various modules and parameter settings.

6.1 Implementation in IrisNet

Our implementation of IDP consists of three modules.

The Stat-Keeper Module. This module runs in each IrisNet OA (*i.e.*, at each host). For each object, we partition its read traffic into *internal* reads, in which the object is read immediately after reading its parent on the same OA, and *external* reads, in which a (sub)query is routed directly to the object from the user query interface or another OA. For each object in an OA's local database, the stat-keeper maintains (1) the object's storage requirements, (2) its external read load, defined as an exponentially-decaying average of the rate of external reads, (3) its internal read load, defined as an exponentially-decaying average of the rate of internal reads, (4) its (external) write load, defined as an exponentially-decaying average of the rate of writes, (5) GNP_{read} , and (6) GNP_{write} . (GNP_{read} and GNP_{write} are defined in Section 4.3.)

Moreover, the stat-keeper maintains the current size, aggregate read load, and aggregate write load for each local database. When an OA with multiple local databases (from multiple co-existing services) incurs excessive load, the local database with maximum size or load sheds its load.

The module is implemented as a light weight thread within the query processing component of the OA. On receiving a query or an update, the relevant statistics (maintained in the main memory) are updated, incurring very negligible overhead ($< 1\%$ CPU load).

The Load Adjusting Module. This module is at the center of IDP's dynamic data placement algorithm. It uses different statistics gathered by the stat-keeper module, and when the OA is overloaded, performs the tasks described in Section 4.1 and Section 4.2.

The OA Discovery Module. This module performs the tasks described in Section 4.3 and acts as a matchmaker between the OAs looking to shed load and the OAs capable of taking extra load.

Currently, the first heuristic described in Section 4.3 (selecting neighboring nodes) is implemented in a distributed way. Each OA exchanges its current load information with its neighboring OAs by piggybacking it with the normal query and response traffic. Thus each OA knows how much extra load each of its neighboring OAs can take, and can decide if any of these OAs can accept the load of a replicated or split fragment. The second heuristic (selecting optimally located nodes), however, is implemented using a centralized directory server where underloaded OAs periodically send their location and load information. Our future work includes implementing this functionality as a distributed service running on IrisNet itself.

6.2 Parameter Settings

We next present the tradeoffs posed by different POST parameters and their settings within our implementation. The values are chosen based on experiments with our implementation subjected to a real workload (more details in Section 7.1 and in [33]).

Watermark Thresholds. The value of the high watermark threshold Th_{high} is set according to the capacity of an OA (e.g., the number of objects per minute that can be processed by queries). The value of the low watermark threshold Th_{low} poses the following tradeoff. A smaller value of Th_{low} increases the data placement cost (and the average response time) due to increased fragmentation of the global database. However, a larger value of Th_{low} causes data placement thrashing, particularly when the workload is bursty. Our experiments in [33] show that a value of $Th_{low} = \frac{4}{5}Th_{high}$ provides a good balance among different degrees of burstiness.

Deletion Threshold. The deletion threshold Th_{del} for an object presents additional trade-offs. A small Th_{del} keeps replicas even if they are not required. This increases the local database size and hence the OA processing time.⁴ A high Th_{del} can remove replicas prematurely, increasing the data placement cost. Based on our experiments, we set the threshold such that an object gets deleted if it receives no queries in the past 30 minutes (and the object has more than the minimum number of replicas).

We use these parameter settings in our experiment evaluation, discussed next.

7 Experimental Evaluation

Our evaluation consists of the following two steps. First, we use a real workload with our deployment of IDP (within IrisNet) running on the PlanetLab to study how IDP's different components work in practice. Then, we use simulation and synthetic workloads to understand how sensitive IDP is to different workloads.

7.1 Performance in Real Deployment

In this section, we evaluate IDP using IrisLog, an IrisNet-based Network Monitor application deployed on 310 PlanetLab hosts. Each PlanetLab host in our deployment runs an IrisNet OA. Computer programs running on each PlanetLab host collect resource usage statistics, and work as sensors to report to the local OA once every 30 seconds. We drive our evaluation by using IrisLogTrace, a real user query trace⁵ collected from our IrisLog deployment on the PlanetLab from 11/2003 to 8/2004 (Table 2). Because

⁴In existing DOM-based XML database engines, query processing time increases with the local database size.

⁵Available at <http://www.intel-iris.net/ilog-trace/>.

Total queries	6467 (100%)
Queries selecting a complete subtree	6409 (99.1%)
Queries selecting all the data	401 (6%)
Queries selecting a country	1215 (19%)
Queries selecting a region	3188 (49%)
Queries selecting a site	1469 (23%)
Queries selecting a host	136 (2%)
Queries not selecting a complete subtree	58 (0.9%)

Table 2: *IrisLogTrace: Trace of user queries from 11/10/2003 to 8/27/2004 for the IrisLog application deployed on 310 PlanetLab hosts.*

the trace is relatively sparse, we replay this trace with a *speedup factor*, defined as the ratio between the original duration of the trace and the replay time. For the experiments requiring longer traces than what we have, we concatenate our trace to construct a longer trace. At the beginning of each experiment, the global database is randomly partitioned among a small group of hosts. As the workload is applied, IDP distributes the database among the available hosts, with the data placement costs decreasing over time. We say that the system has reached *initial steady state* if, under our reference workload, no data relocation occurs within a period of 2 hours. We experimentally ensured that this choice of period was reasonable. In all our experiments, we measure the cost of data placement from the very beginning of the experiment, while all other metrics (e.g., response time) are measured after the system has reached its initial steady state, at which point the specific workload to be studied is started.

Figure 1 demonstrates the overall performance of IDP. As shown, IDP helps IrisNet maintain reasonable response times even under a high flash-crowd. Without IDP, IrisNet becomes overloaded and results in high response times and timeouts.

We now evaluate the individual components of IDP.

7.1.1 Partitioning Heuristics

This section evaluates POST and compares it with three hypothetical algorithms: GREEDY, LOCALOPT and ORACLE. In GREEDY, overloaded hosts evict individual objects in decreasing order of their loads. As a result, hosts make decisions with finer granularity, but do not try to keep the hierarchical objects clustered. In LOCALOPT, each host partitions its XML fragment using an optimal tree partitioning algorithm [30] with $O(n^3)$ complexity. However, because each invocation of LOCALOPT takes tens of minutes of computation time, we do not use it in our live experiments on the PlanetLab. ORACLE is an offline approach that takes the entire XML database and the query workload and computes the optimal fragmentation (again, using the algorithm in [30]). ORACLE cannot be used in a real system and only serves as a lower bound for comparison purposes.

Our evaluation consists of two phases. In Phase 1, we

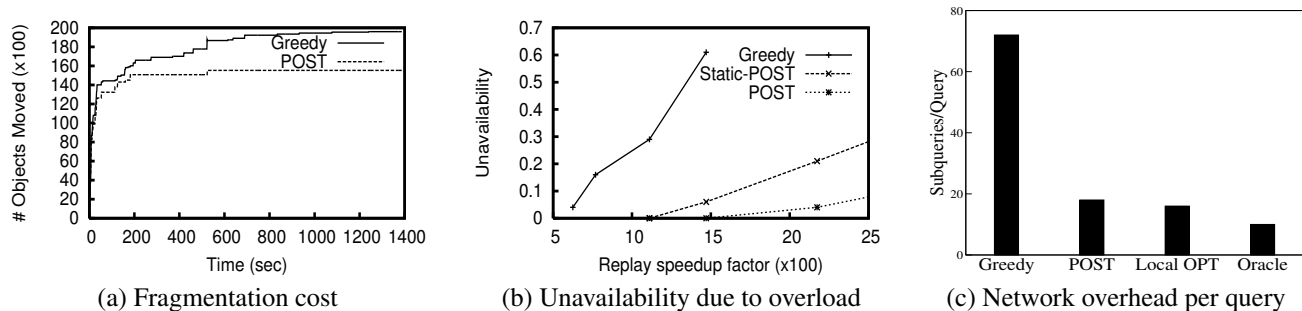


Figure 5: Comparing different fragmentation algorithms.

fragment the IrisLog database by using all but the last 1000 queries of IrisLogTrace as warm-up data. To do this within a reasonable amount of time, we perform this phase on the Emulab testbed [6] on a LAN. This enables us to use a large speedup factor of 1200 and finish this phase in a short period of time. We also assign each Emulab host its GNP coordinates that can be used to model the latency between any two emulated hosts if required.

In Phase 2 of our experiment, we place the fragments created in Phase 1 in our PlanetLab deployment and inject the last 1000 queries from IrisLogTrace.

To understand the advantage of POST’s adaptive fragmentation, we also use Static-POST which, under all speedup factors, starts from the same warm-up data (generated from Phase 1 with a speedup factor of 1200) and does not further fragment the database during Phase 2.

Figure 5(a) plots the cumulative overhead of the fragmentation algorithms over time during the warm-up phase of the experiment. The graph shows that the amount of fragmentation decreases over time, which means that our load-based invocations of the algorithms do converge under this workload. GREEDY incurs higher overhead than POST because GREEDY’s non-clustering fragmentation increases the overall system load which makes the hosts fragment more often. We do not use LOCALOPT or ORACLE in this experiment due to their excessive computation overhead.

Replaying IrisLogTrace with a high speedup factor on IrisLog overloads the hosts. Because of this overload, the response of a typical query may return only a fraction of the requested objects. The remaining objects reside in overloaded hosts that fail to respond to a query within a timeout period (default 8 seconds in IrisNet). We define the unavailability of a query to be the fraction of the requested objects not returned in the answer. Figure 5(b) shows the average unavailability of IrisLog under different fragmentation algorithms and under different speedup factors. GREEDY is very sensitive to load and suffers from high unavailability even under relatively smaller speedup factors (*i.e.*, smaller load). This is because GREEDY produces suboptimal fragments and overloads the system by generating a large number of subqueries per query (Fig-

ure 5(c)). POST is significantly more robust and it returns most objects even at a higher speedup factor. The effectiveness of POST comes from its superior choice of fragments, which generate a near optimal number of subqueries (as shown in Figure 5(c)). The difference between Static-POST and POST in Figure 5(b) demonstrates the importance of adaptive load-shedding for better robustness against a flash-crowd-like event.

7.1.2 Data Placement Heuristics

We now evaluate the effectiveness of different heuristics described in Section 4.3 for choosing a target host for a fragment. The heuristics are compared to the naive approach of choosing any random host that can accept the extra load. We start this experiment by placing the fragments generated by Phase 1 of the previous experiment and then replaying the last 1000 queries from IrisLogTrace with a moderate speedup factor. In different settings, we use a different combination of placement heuristics. We represent the cost of a data transfer using the metric *object-sec*, which is the product of the number of objects transferred and the latency between the source and the destination. Intuitively, this metric captures both the amount of data transferred and the distance over which it is transferred. The results of this experiment are shown in Figure 6. As mentioned before, response times and read/write traffic are computed only in the second phase of the experiment (done on the PlanetLab), while the data placement cost includes the cost incurred in both phases.

The first heuristic *Opt Loc*, choosing a host near the optimal location, reduces network latency and traffic, as shown in Figure 6. The second heuristic *Clustering*, choosing a neighboring host, reduces the number of hops between hosts. This is due to the fact that the heuristic increases the average cluster size (25% increase on average in our results). To achieve the best of both heuristics, we use them together as follows. Because host processing time dominates network latency, we first try to find a neighboring host capable of taking the extra load; only if no such host exists, we use the second heuristic to find a host near the optimal location. This has the following interesting implication: because the read and write sources

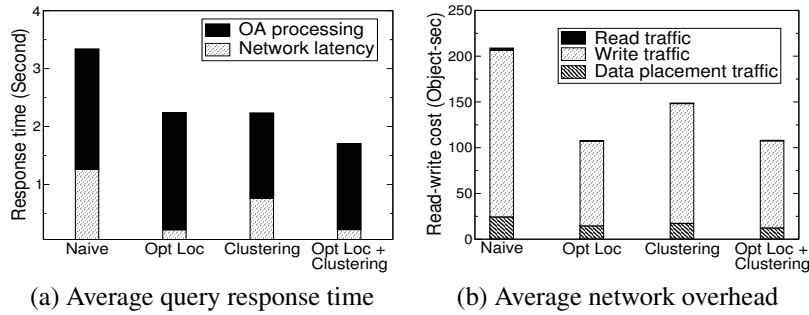


Figure 6: Effect of data placement heuristics on query response time and network overhead.

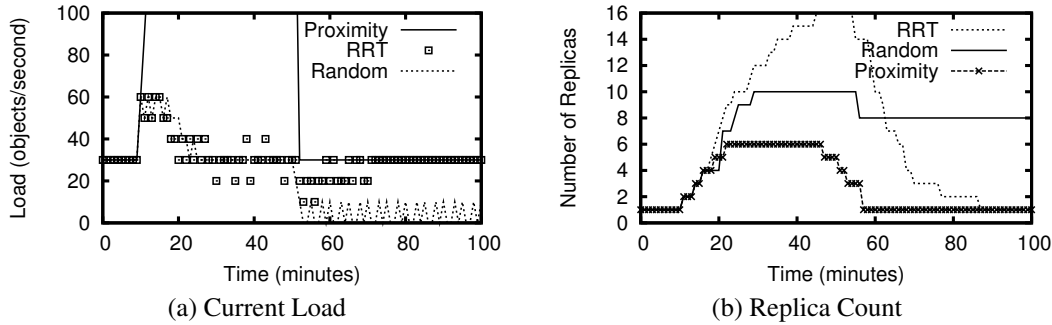


Figure 7: Change in load of a host and replica count of a fragment after a burst of read traffic.

of the objects that are adjacent in the global database are expected to be nearby (because of read-write locality of sensor data), first placing an object in the optimal host and then placing the adjacent objects in the same host automatically places the objects near the optimal location. As shown in the graph, the combination of the two heuristics reduces the total response time and read network traffic, with a slight increase in write network traffic. Also note that the data placement cost is many times smaller than the read/write cost, showing that IDP has little overhead.

7.1.3 Replica Selection Heuristics

To demonstrate the effect of different replica selection heuristics, we do the following experiment. We place a fragment in a host H on the PlanetLab. We start issuing queries from a nearby location so that H is the optimal host for the reader. At time $t < 10$, the load of the host is below the threshold (≈ 35 , the maximum load H can handle). At $t = 10$, we increase the query rate by a factor of 10. This flash crowd exists until $t = 50$, after which the read load drops below the threshold. As the query rate increases, the host creates more and more replicas. In different versions of this experiment, we use different heuristics to select a replica to which the query is sent. We plot the load of the host H and the number of replicas at different times in Figure 7.

With *proximity-based replica* selection, host H becomes overloaded and creates new replicas. However, as expected, all the load shedding efforts are in vain because

all the queries continue to go to H . As the figure shows, its current load promptly goes up (the peak value 400 is not shown in the graph) while all the newly created replicas remain unused and eventually get deleted.

With *random* selection, the load is uniformly distributed among the replicas. As shown in Figure 7, as the load increases, fragments are replicated among the hosts until the load of each host goes below the threshold ($t = 20$). However, after the flash crowd disappears ($t = 50$), all the replicas still get the same equal fraction of the total load. In this particular experiment, at $t = 51$, the average load of each replica is slightly below the threshold used for deletion of objects. Because of the randomized deletion decision, two replicas get deleted at $t = 57$. This raises the average load of each replica above the deletion threshold. As a result, these replicas remain in the system and consume additional resources (Figure 7(b)).

Randomized response-time (RRT) selection shows the same good load balancing properties of random selection. Moreover, after the flash crowd disappears, the replica at the optimal location gets most of the read load. Thus, the other replicas become underloaded and eventually get deleted. Furthermore, because RRT incorporates estimated response times of replicas in its replica selection, it significantly reduces the query response time (by around 25% and 10% compared to proximity-based and random selection, respectively). This demonstrates the effectiveness of RRT for sensing services.

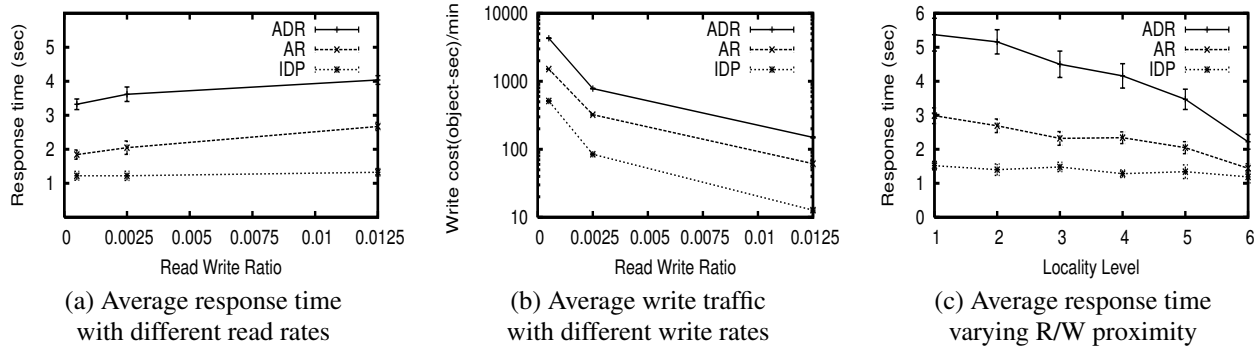


Figure 8: Effect of varying the Read rate, the Write rate, and the R/W proximity.

7.2 Sensitivity Analysis with Simulation

We now use simulation to understand how IDP performs as the different parameters of the workload change. Our IrisNet simulator enables us to use a larger setup (with 10,000 hosts) than our PlanetLab deployment of IrisLog. Each simulated host is given multi-dimensional coordinates based on measurements from the GNP [34] network mapping system, and the cartesian distance between hosts estimates the latency between them. We use the GNP coordinates of the 867 hosts from [34]. Because the simulated hosts and sensors outnumber the available GNP coordinates by two orders of magnitude, we emulate additional GNP coordinates by adding small random noise to the actual GNP coordinates. To approximate the geographic locality of sensors, we make sure that sensors that report to sibling leaf objects are assigned nearby GNP coordinates (*e.g.*, the sensors that report to block 1, 2, and 3 of Oakland have nearby coordinates). The simulator does not model the available bandwidth between hosts in the network. However, most messages are small and, therefore, their transmission times will be limited more by latency than by bandwidth. Unless otherwise stated, we use IrisLogTrace with an appropriate speedup factor for our read workload. For our write workload, we assume that every sensor reports data once every 30 seconds.

We compare IDP with the following existing adaptive data placement approaches. In the first approach, *aggressive replication* (AR), a host maintains replicas of the objects it has read from other hosts. The second approach, *adaptive data replication* (ADR), uses the algorithm proposed in [41]. Although ADR provably minimizes the amount of replication compared to aggressive replication, it incurs a higher communication cost. This is because, for general topologies, it requires building a spanning tree of the hosts and communicating only through that tree, which is crucial for the optimality of its data placement overhead. In all the schemes, if a host has insufficient capacity to store the new objects, it replaces a randomly-chosen least-recently-used object.

Note that comparing IDP with ADR or AR is not an

apples-to-apples comparison, since ADR and AR were designed for different workloads. Still, our experiments provide important insight about their performance under a sensing workload. For lack of space, we present only a few interesting results. More details of the simulation setup and additional results can be found in [33].

Read and Write Rate. Figure 8(a) shows that for a given write rate (16,000 objects/min),⁶ the average query response time of IDP increases slightly with the read rate. This is because of the increased fragmentation due to higher system load. Surprisingly, even with AR, the response time increases with the read rate. The reason behind this counter-intuitive performance is that AR is oblivious to data clustering. ADR has the worst response time because it communicates over a spanning tree, ignores data clustering, and does not consider the hierarchical data access patterns. At all the read rates shown in Figure 8(a), IDP performs significantly better than the other algorithms. Similarly, our experiments have shown that IDP incurs significantly less read traffic than these algorithms across all the read rates.

In Figure 8(b), we vary the write rate with a constant read rate (40 objects/min). As expected, the write cost increases as the write rate increases. Although both ADR and IDP create similar numbers of replicas, ADR incurs a higher write cost because updates are propagated using a spanning tree overlay and, therefore, they traverse longer distances. AR incurs more cost than IDP, because it fails to capture the locations of write sources while placing replicas.

Read-Write Proximity. Read-write proximity reflects the physical distance between the reader and the writer. As mentioned in Section 2.1, we model it by the lowest level (called a locality level) in the hierarchy they have in common. Our simulated database has 6 levels, with the root at level 1. Thus, a locality level of 1 signifies the smallest proximity (because the reader and the writer for

⁶To be more accurate, we here use the number of objects touched, instead of the number of updates or queries per minute.

the same data can be anywhere in the whole hierarchy), and a level of 6 means the highest proximity.

Figure 8(c) shows that increasing the locality level reduces the average response time. This is intuitive, because queries with higher levels of locality select fewer objects, and, thus, need to traverse fewer hosts. Because it keeps objects clustered, IDP outperforms the other algorithms. The difference in performance becomes more pronounced as the locality level is low, because such queries select large subtrees from the global database and so the advantages of clustering are amplified.

Representative Services Workload. We have also evaluated IDP, AR, and ADR with a wide variety of synthetic workloads and a heterogeneous mix of them (more details in [33]). By tuning different parameters, we have generated workloads outlined in Table 1. Our results show that IDP outperforms the other two algorithms in both response time and network overhead for all the workloads. AR has worse response time because it ignores data clustering and optimal placement, and worse network overhead because it creates too many replicas. The performance of AR approaches that of IDP when both (1) the write rate is low (*e.g.*, Epidemic Alert) so that eager replication has minimal cost, and (2) the read locality is near the leaf objects (*e.g.*, Parking Finder) so that typical queries select only small subtrees and hence clustering objects is not that crucial. On the other hand, ADR has worse response time and network overhead for all the services, mainly because it communicates over a spanning tree, ignores data clustering, and does not consider the hierarchical data access patterns.

8 Related Work

The issues of data replication and placement in wide-area networks have been studied in a variety of contexts. In this section, we first discuss the relevant theoretical analyses and then present relevant efforts in Web content delivery, distributed file systems, and distributed databases.

Theoretical Background. The off-line data allocation problem (*i.e.*, the placement of data replicas in a system to reduce a certain cost function) has been studied extensively (see [15] for a survey). The general problem has been found to be NP-Hard [42], and several approximate and practical solutions have been proposed [21, 28, 36]. The similar problem of optimally replicating read-only objects has been addressed in the context of content distribution networks [22] and the Web [35].

Other studies [11, 29, 41] have explored the on-line replication of objects in distributed systems. The competitive algorithm in [11] is theoretically interesting, but has little practical application since on every write, all but one replica is deleted. [29] uses expensive genetic algorithms to approximate optimal replication, and requires global information (*e.g.*, current global read and write loads).

The ADR algorithm in [41] provides optimal data placement when the network topology is a tree. However, its performance on a general topology is worse than IDP, as shown in Section 7.

Web Content Delivery. Web proxy caches [40] replicate data on demand to improve response time and availability, but deal with relatively infrequent writes and simpler access patterns. The replication schemes used by distributed cache systems and content distribution systems (CDNs) [1, 23] place tighter controls on replication in order to manage space resources even more efficiently and to reduce the overhead of fetching objects. These approaches may be more applicable to read/write workloads. However, they do not support the variable access granularity and frequent writes of sensing services.

Distributed File Systems. Some recent wide-area distributed file systems have targeted supporting relatively frequent writes and providing a consistent view to all users. The Ivy system [32] provides support for the combination of wide-area deployment and write updates. However, while creating replicas in the underlying distributed hash table (DHT) is straightforward, controlling their placement is difficult. The Pangaea system [36] uses aggressive replication along with techniques (efficient broadcast, harbingers, *etc.*) for minimizing the overhead of propagating updates to all the replicas. However, our experiments in Section 7 show that the less aggressive replication of IDP provides better read performance while significantly reducing write propagation overhead. These existing works do not consider the complex access patterns and dynamic access granularity that we consider.

Distributed Databases. Distributed databases provide similar guarantees as file systems while supporting more complex access patterns. Off-line approaches [10, 24] to the replication problem require the complete query schedule (workload) *a priori*, and determine how to fragment the database and where to place the fragments. However, neither [10] nor [24] considers the storage and processing capacity of the physical sites, and [24] considers only read-only databases. Brunstrom et al. [12] considers data relocation under dynamic workloads, but assumes that the objects to be allocated are known *a priori* and does not consider object replication. It also allocates each object independently and thus fails to exploit the possible access correlation among objects. The PIER system [19], like Ivy, relies on an underlying DHT for replication of data. The Mariposa system [37] considers an economic model for controlling both the replication of table fragments and the execution of queries across the fragmented database. While Mariposa's mechanisms are flexible and could create arbitrary replication policies, the work does not evaluate the policies. All of these past efforts are very specific to relational databases, and, therefore, are not directly applicable to the hierarchical database systems that we explore.

9 Conclusion

In this paper, we identified the unique workload characteristics of wide-area sensing services, and designed the IDP algorithm to provide automatic, adaptive data placement for such services. Key features of IDP include proactive coalescing, size-constrained optimal-partitioning, TCP-time-out-inspired load estimators, GNP-based latency estimators, parent-child peering soft-state fault tolerance, placement strategies that are both cluster-forming and latency-reducing, and randomized response-time replica selection. We showed that IDP outperforms previous algorithms in terms of response time, network traffic, and responsiveness to flash crowds, across a wide variety of sensing service workloads.

Acknowledgements. We thank Brad Karp and Mahadev Satyanarayanan for helpful discussions on this work.

References

- [1] Akamai web site. <http://www.akamai.com/>.
- [2] Argus. <http://www.netcoast.nl/info/argus/argus.htm>.
- [3] Biomedical security institute. <http://www.biomedsecurity.org/info.htm>.
- [4] Cisco IOS NetFlow - Cisco Systems. http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html.
- [5] Collaborative adaptive sensing of the atmosphere. <http://www.casa.umass.edu/>.
- [6] Emulab - network emulation testbed home. <http://www.emulab.net>.
- [7] IrisLog: A distributed syslog. <http://www.intel-iris.net/irislog.php>.
- [8] IrisNet: Internet-scale resource-intensive sensor network services. <http://www.intel-iris.net/>.
- [9] Trends in RFID technology at Walmart. <http://www.rfida.com/walmartfrid.htm>.
- [10] APERS, P. M. G. Data allocation in distributed database systems. *ACM Trans. on Database Systems* 13, 3 (1988).
- [11] AWERBUCH, B., BARTAL, Y., AND FIAT, A. Optimally-competitive distributed file allocation. In *ACM STOC* (1993).
- [12] BRUNSTROM, A., LEUTENEGGER, S. T., AND SIMHA, R. Experimental evaluation of dynamic data allocation strategies in a distributed database with changing workloads. In *4th Int. Conf. on Information and Knowledge Management* (1995).
- [13] DAHLIN, M. Interpreting stale load information. *IEEE Trans. on Parallel and Distributed Systems* 11, 10 (2000).
- [14] DESHPANDE, A., NATH, S., GIBBONS, P. B., AND SESHAN, S. Cache-and-query for wide area sensor databases. In *ACM SIGMOD* (2003).
- [15] DOWDY, L., AND FOSTER, D. Comparative models of the file assignment problem. *ACM Computing Surveys* 14, 2 (1982).
- [16] FELDMANN, A., GREENBERG, A., LUND, C., REINGOLD, N., REXFORD, J., AND TRUE, F. Deriving traffic demands for operational IP networks: Methodology and experience. *IEEE/ACM Trans. on Networking* 9, 3 (2001).
- [17] FRANKLIN, M. J., JEFFERY, S. R., KRISHNAMURTHY, S., REISS, F., RIZVI, S., WU, E., COOPER, O., EDAKKUNNI, A., AND HONG, W. Design considerations for high fan-in systems: The HiFi approach. In *CIDR* (2005).
- [18] GIBBONS, P. B., KARP, B., KE, Y., NATH, S., AND SESHAN, S. IrisNet: An architecture for a worldwide sensor web. *IEEE Pervasive Computing* 2, 4 (2003).
- [19] HUEBSCH, R., HELLERSTEIN, J. M., LANHAM, N., LOO, B. T., SHENKER, S., AND STOICA, I. Querying the internet with PIER. In *VLDB* (2003).
- [20] JACOBSON, V. Congestion avoidance and control. In *ACM SIGCOMM* (1988).
- [21] KALPAKIS, K., DASGUPTA, K., AND WOLFSON, O. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. on Parallel and Distributed Systems* 12, 6 (2001).
- [22] KANGASHARJU, J., ROBERTS, J., AND ROSS, K. Object replication strategies in content distribution networks. In *Int. Web Caching and Content Distribution Workshop* (2001).
- [23] KARGER, D., SHERMAN, A., BERKHEIMER, A., BOGSTAD, B., DHANIDINA, R., IWAMOTO, K., KIM, B., MATKINS, L., AND YERUSHALMI, Y. Web caching with consistent hashing. *Computer Networks* 31, 11-16 (1999).
- [24] KARLAPEL, K., AND PUN, N. M. Query-driven data allocation algorithms for distributed database systems. In *Database and Expert Systems Applications* (1997).
- [25] KARYPIS, G., AND KUMAR, V. A fast and high quality multi-level scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998).
- [26] KUMAR, P. R. Information processing, architecture, and abstractions in sensor networks. SenSys'04 Invited Talk.
- [27] KUROSE, J. Collaborative adaptive sensing of the atmosphere. SenSys'04 Invited Talk. <http://www.casa.umass.edu/>.
- [28] LAM, K., AND YU, C. T. An approximation algorithm for a file-allocation problem in a hierarchical distributed system. In *ACM SIGMOD* (1980).
- [29] LOUKOPOULOS, T., AND AHMAD, I. Static and adaptive data replication algorithms for fast information access in large distributed systems. In *ICDCS* (2000).
- [30] LUKES, J. A. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development* 18, 3 (1974).
- [31] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. Tag: A tiny aggregation service for ad hoc sensor networks. In *OSDI* (2002).
- [32] MUTHITACHAROEN, A., MORRIS, R., GIL, T. M., AND CHEN, B. Ivy: A read/write peer-to-peer file system. In *SOSP* (2002).
- [33] NATH, S., GIBBONS, P. B., KARP, B., AND SESHAN, S. Adaptive data placement for wide-area sensing services. Tech. Rep. IRP-TR-04-05, Intel Research Pittsburgh, October 2004.
- [34] NG, T. S. E., AND ZHANG, H. Predicting internet network distance with coordinates-based approaches. In *INFOCOM* (2002).
- [35] QIU, L., PADMANABHAN, V. N., AND VOELKER, G. M. On the placement of web server replicas. In *INFOCOM* (2001).
- [36] SAITO, Y., KARAMANOLIS, C., KARLSSON, M., AND MAHALINGAM, M. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI* (2002).
- [37] SIDELL, J., AOKI, P. M., SAH, A., STAELIN, C., STONEBRAKER, M., AND YU, A. Data replication in Mariposa. In *ICDE* (1996).
- [38] SZE, C. N., AND WANG, T.-C. Optimal circuit clustering for delay minimization under a more general delay model. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 22, 5 (2003).
- [39] VAN RENESSE, R., BIRMAN, K. P., AND VOGELS, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. on Computer Systems* 21, 2 (2003).
- [40] WANG, J. A survey of Web caching schemes for the Internet. *ACM Computer Communication Review* 25, 9 (1999).
- [41] WOLFSON, O., JAJODIA, S., AND HUANG, Y. An adaptive data replication algorithm. *ACM Trans. on Database Systems* 22, 2 (1997).
- [42] WOLFSON, O., AND MILO, A. The multicast policy and its relationship to replicated data placement. *ACM Trans. on Database Systems* 16, 1 (1991).
- [43] WWWC : THE WORLD WIDE WEB CONSORTIUM. XML Path Language (XPATH). <http://www.w3.org/TR/xpath>, 1999.

Ursa Minor: versatile cluster-based storage

Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger,
James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad,
Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen,
John D. Strunk, Eno Thereska, Matthew Wachs, Jay J. Wylie
Carnegie Mellon University

Abstract

No single encoding scheme or fault model is optimal for all data. A versatile storage system allows them to be matched to access patterns, reliability requirements, and cost goals on a per-data item basis. Ursa Minor is a cluster-based storage system that allows data-specific selection of, and on-line changes to, encoding schemes and fault models. Thus, different data types can share a scalable storage infrastructure and still enjoy specialized choices, rather than suffering from “one size fits all.” Experiments with Ursa Minor show performance benefits of $2\text{--}3\times$ when using specialized choices as opposed to a single, more general, configuration. Experiments also show that a single cluster supporting multiple workloads simultaneously is much more efficient when the choices are specialized for each distribution rather than forced to use a “one size fits all” configuration. When using the specialized distributions, aggregate cluster throughput nearly doubled.

1 Introduction

Today’s enterprise storage is dominated by large monolithic disk array systems, extensively engineered to provide high reliability and performance in a single system. However, this approach comes with significant expense and introduces scalability problems, because any given storage enclosure has an upper bound on how many disks it can support. To reduce costs and provide scalability, many are pursuing cluster-based storage solutions (e.g., [2, 8, 9, 10, 11, 12, 18, 23]). Cluster-based storage replaces the single system with a collection of smaller, lower-performance, less-reliable storage-nodes (sometimes referred to as *storage bricks*). Data and work are redundantly distributed among the bricks to achieve higher performance and reliability. The argument for the cluster-based approach to storage follows from both the original RAID argument [27] and arguments for cluster computing over monolithic supercomputing.

Cluster-based storage has scalability and cost advantages, but most designs lack the versatility commonly

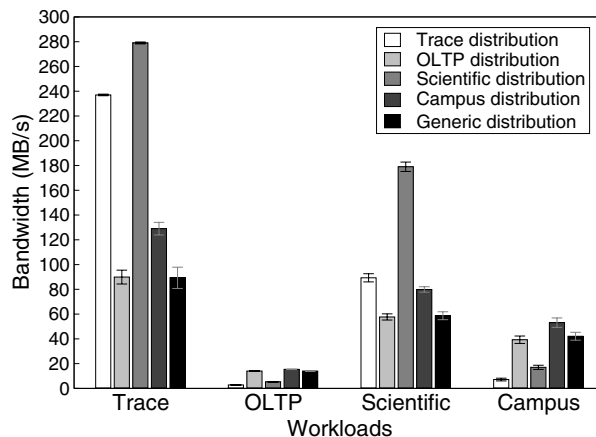


Figure 1: Matching data distribution to workload. This graph shows the performance of four workloads run on Ursa Minor as a function of the data distribution. For each workload, five distributions were evaluated: the best distribution for each of the four workloads and a generic “middle of the road” choice for the collection of workloads. Although the “Scientific” data distribution provided better performance for the “Trace” workload than the “Trace” distribution, and the “Campus” data distribution provided better performance for the “OLTP” workload than the “OLTP” distribution, these distributions failed to meet the respective workloads’ reliability requirements. Section 4.3 details the workloads and data distributions. These numbers are the average of 10 trials, and the standard deviations are shown as error bars.

found in high-end storage solutions. By *versatility*, we mean that first-order data distribution choices (e.g., data encoding, fault tolerance, and data location) can be specialized to individual data stored within a single infrastructure. Such versatility is crucial for addressing the varied demands of different classes of data. Failing to provide versatility forces all data into a single point of the performance/reliability/cost trade-off space. Versatility also addresses the impact of access patterns on the performance of different data distributions. For example, data accessed with large, sequential I/Os often should be erasure coded to reduce the capacity and bandwidth costs of fault tolerance, while randomly-accessed data often should be replicated to minimize the number of disk I/Os per data access.

This paper describes Ursa Minor, a versatile, cluster-based storage system designed to allow the selection of, as well as on-line changes to, the data location, encoding, block size, and fault model on a per-“data object” basis. Ursa Minor achieves its versatility by using a protocol family, storing variably-sized data-fragments at individual storage-nodes, and maintaining per-object data distribution descriptions. Using a protocol family shifts the decision of which types of faults to mask from system implementation time to data creation time. This allows each object within a single infrastructure to be protected from the types and quantities of faults appropriate to that particular class of data. Ursa Minor’s protocol family supports a per-object choice of *data distribution*. This includes the data encoding (replication or erasure coding), block size, storage-node fault type (crash or Byzantine), number of storage-node faults to tolerate, timing model (synchronous or asynchronous), and data location. Storage-nodes treat all objects similarly, regardless of the object’s data distribution.

Experiments with our implementation of Ursa Minor validate both the importance of versatility and Ursa Minor’s ability to provide it. As illustrated in Figure 1, significant performance benefits are realized when the data distribution choice is specialized to access patterns and fault tolerance requirements. These benefits remain even when multiple workload types share a storage cluster. In addition to performance benefits, capacity benefits are also realized when erasure coding is used instead of replication. For example, the data distribution for the Trace workload uses erasure coding to reduce space consumption by 50% while tolerating two crash failures; only a 10% performance penalty is paid for doing this, because the workload is highly sequential. Similarly, specializing the fault model ensures that costs for fault tolerance are incurred in accordance with acceptable risks, increasing throughput for data with lesser reliability requirements (e.g., the Scientific workload) by as much as a factor of three over a reasonable “one size fits all” configuration.

Ursa Minor’s ability to support on-line data distribution change is also demonstrated. The ability to reconfigure data distributions on-line enables tuning based on observed usage rather than expected usage. This simplifies tuning, since pre-deployment expertise about an application’s access patterns becomes less important. Minimizing the amount of pre-deployment expertise and planning is important for reducing the excessive administration effort required with today’s storage infrastructures. Additionally, the ability to make on-line distribution changes allows the system to be adapted as goals and workloads evolve.

This paper makes the following contributions. First, it makes a case for versatile cluster-based storage, demon-

strating that versatility is needed to avoid significant performance, reliability, and/or cost penalties when storage is shared among different classes of data. Second, it describes the design and implementation of Ursa Minor, a versatile, cluster-based storage system. We are aware of no existing cluster-based storage system that provides nearly as much versatility, including the ability to specialize fault models and to change data distributions on-line. Third, it presents measurement results of the Ursa Minor prototype that demonstrate the value of specializing according to access patterns and reliability requirements as well as the value of allowing on-line changes to data distributions.

2 Versatile cluster-based storage

Today’s enterprise storage systems are typically monolithic and very expensive, based on special-purpose, high-availability components with comprehensive internal redundancy. These systems are engineered and tested to tolerate harsh physical conditions and continue operating under almost any circumstance. They provide high-performance and high-reliability, but they do so at great monetary expense.

Cluster-based storage is a promising alternative to today’s monolithic storage systems. The concept is that collections of smaller storage-nodes should be able to provide performance and reliability competitive with today’s high-end solutions, but at much lower cost and with greater scalability. The cost reductions would come from using commodity components for each storage-node and exploiting economies of scale. Each storage-node would provide a small amount of the performance needed and lower reliability than required. As with previous arguments for RAID and cluster computing, the case for cluster-based storage anticipates that high levels of reliability and performance can be obtained by appropriate redundancy and workload distribution across storage-nodes. If successful, cluster-based storage should be much less expensive (per terabyte) than today’s enterprise storage systems, while providing similar levels of reliability and availability [10].

Cluster-based storage also helps with the scaling challenges inherent in monolithic storage systems. In particular, once the limit on the number of disks that can be inserted into a large storage system’s enclosures is reached, a second large system must be purchased and data must be redistributed across the systems. Avoiding this drastic step-function in effort and capital expenditure can push administrators to purchase oversized (but mostly empty) systems. Most cluster-based storage designs allow growth of capacity and performance through the in-

cremental addition of storage-nodes, with automated balancing of the data to utilize the new resources.

2.1 Versatility in cluster-based storage

To replace monolithic storage effectively, cluster-based storage must provide similar versatility. It must be possible to specialize the data distribution for different classes of data and their respective workloads.

This section describes several choices that should be specialized to individual data based on application requirements (e.g., fault tolerance and performance goals), access patterns, and cost restrictions. Almost all modern disk array systems allow the encoding scheme (e.g., RAID 5 vs. RAID 0 + 1) and stripe unit size to be chosen on a per-volume basis. Cluster-based systems should have similar versatility. In addition, cluster-based storage introduces questions of fault model choice that have a greater impact than in the centralized controller architecture of monolithic storage systems.

Data encoding choices: Data can be spread across cluster storage-nodes to address two primary concerns: fault tolerance and load balancing. In most cluster-based storage designs, assignment of data to storage-nodes is dynamically adapted to balance load. The approach to fault tolerance, on the other hand, is often fixed for all data.

There are two common encoding schemes for cluster-based storage. First, data can be *replicated* such that each block is stored on two or more storage-nodes. Second, data can be *erasure coded*. For example, an m -of- n erasure code encodes a data block into n fragments such that any m can be used to reconstruct it.¹ The trade-off between these schemes is similar to that of RAID 5 versus RAID 0 + 1 in disk array systems. Replicated data generally supports higher disk-bound throughput for non-sequential accesses. On the other hand, erasure coded data can tolerate failures (especially multiple failures) with less network bandwidth and storage space [40, 43]. For sequentially accessed data, these benefits can be realized without significant disk access penalties.

Most modern disk array systems use data distributions that can tolerate a single disk failure. This is unlikely to be sufficient in cluster-based storage systems that use less robust components than traditional systems. Further, other components (e.g., fans and power supplies) that can fail and be hot-swapped in high-end storage systems will translate into storage-node failures in cluster-based storage systems. The expectation, therefore, is more frequent storage-node failures. Even with traditional systems, manufacturers have recognized the importance of tolerating multiple disk failures [6]. In cluster-based stor-

age, tolerating two or more storage-node failures is likely to be required for important data. Because of the performance and capacity tradeoffs, however, the number of failures tolerated must be configurable.

Fault model choices: In traditional systems, a centralized controller provides a serialization point, single restart location, and an unambiguous storage-node (i.e., disk) failure indication. In contrast, most cluster-based storage designs are decentralized systems, enjoying none of these luxuries. As a result, carefully designed data access protocols are utilized to provide data consistency in the face of storage-node failures, communication delays, client failures, and concurrent access.

The overheads associated with these protocols depend significantly on their underlying fault model assumptions, and there are many choices. For example, one might assume that faulty storage-nodes only ever crash or that they might behave more arbitrarily (e.g., corrupting data or otherwise not cooperating). One might assume that clocks are synchronized and communication delays are bounded (i.e., a *synchronous* timing model) or that storage-node reboots and transient network delays/partitions make timing assumptions unsafe (i.e., an *asynchronous* timing model). Weakening failure and timing assumptions generally make a system more robust at the expense of additional data redundancy and communication.

It is tempting to assume that tolerating storage-node crashes is sufficient and that good engineering can prevent Byzantine (i.e., non-crash) failures and timing faults. However, given the amount of software involved and the consumer-quality components that are likely to be integrated into cluster-based storage systems, there is significant risk associated with that assumption. Even in today's high-end storage systems, there are mechanisms designed to mask non-crash communication and firmware failures within the controller and the disks. For example, we have been told [20] that disks occasionally write data sectors to the wrong location.² Such a fault corrupts two pieces of data: the old version of the data goes unmodified (an "omission failure") and some unas- sociated data is replaced. Non-crash failures can be expected to increase in frequency when using less robust components to construct a system.

Ability to change choices on-line: Most cluster-based storage designs adaptively modify the assignments of data replicas/fragments to storage-nodes based on access patterns and storage-node availability. We believe that it is desirable for other data distribution choices to be

¹RAID 5 is an $(n - 1)$ -of- n scheme.

²Exact reasons for this sort of problem are rarely reported, but the observed behavior is not limited to a single disk make or model. It could be caused by bugs in firmware or by hardware glitches induced by vibration, heat, or other physical effects.

adaptable as well. If modifying such choices were easy, administrators could worry less about getting the initial configuration choice perfect, especially with regards to tuning to match access patterns. Instead, applications and their storage could be deployed, and the data distribution choices could be adjusted based on the actual access pattern. Even the number and type of faults tolerated could be changed based on the problems observed in practice.³

By allowing changes based on observed behavior, a system can save storage administrators from having to gain expertise in the impacts of each physical environment and the storage behavior of each major application before deploying a storage infrastructure. Instead, a trial-and-error approach could be used to arrive at an acceptable system configuration. Additionally, on-line change can be invaluable as access patterns and goals change over the course of the data's lifecycle.

2.2 Related work

There is a large body of previous work in cluster-based storage and in adaptive storage systems. This section overviews some high-level relationships to Ursa Minor's goal of versatile cluster-based storage. Related work for specific mechanisms are discussed with those mechanisms.

Many scalable cluster-based storage systems have been developed over the years. Petal [23], xFS [2], and NASD [13] are early systems that laid the groundwork for today's cluster-based storage designs, including Ursa Minor's. More recent examples include FARSITE [1], FAB [34], EMC's Centera [8], EqualLogic's PS series product [9], Lustre [24], Panasas' ActiveScale Storage Cluster [26], and the Google file system [12]. All of these systems provide the incremental scalability benefits of cluster-based storage, as well as some provisions for fault tolerance and load balancing. However, each of them hard-codes most data distribution choices for all data stored in the system. For example, Petal replicates data for fault tolerance, tolerates only server crashes (i.e., fail-stop storage-nodes), and uses chained declustering to spread data and load across nodes in the cluster; these choices apply to all data. xFS also uses one choice for the entire system: parity-based fault tolerance for server crashes and data striping for load spreading. Ursa Minor's design builds on previous cluster-based storage systems to provide versatility. Its single design and implementation supports a wide variety of data distribution

choices, including encoding scheme, fault model, and timing model. All are selectable and changeable on-line on a per-object basis.

FAB [34] and RepStore [45] offer two encoding scheme choices (replication or erasure coding) rather than just one. FAB allows the choice to be made on a per-volume basis at volume creation time. RepStore, which has been designed and simulated, uses AutoRAID-like [41] algorithms to adaptively select which to use for which data. Reported experiments with the FAB implementation and the RepStore simulator confirm our experiences regarding the value of this one form of versatility. Ursa Minor goes beyond both in supporting a much broader range of configuration choices for stored data, including fault models that handle non-crash failures. Compared to FAB, Ursa Minor also supports re-encoding of data, allowing configuration choices to be modified on-line.

Pond [30] uses both replication and erasure coding for data in an effort to provide Internet-scale storage with long-term durability. It uses replication for active access and erasure coding for long-term archiving. Although it does provide incremental scalability, it is designed for wide-area deployment rather than single-data-center cluster-based storage. Partly as a consequence, it does not provide most of the versatility options of Ursa Minor.

An alternative approach to cluster-based storage is to provide scalability by interposing a proxy [35], such as Mirage [3], Cuckoo [21], or Anypoint [44]. Proxies can spread data and requests across servers like a disk array controller does with its disks. This approach to building a storage infrastructure represents a middle-ground between traditional and cluster-based storage.

AutoRAID [41] automates versatile storage in a monolithic disk array controller. Most disk array controllers allow specialized choices to be made for each volume. AutoRAID goes beyond this by internally and automatically adapting the choice for a data block (between RAID 5 and mirroring) based on usage patterns. By doing so, it can achieve many of the benefits from both encodings: the cost-effectiveness of RAID 5 storage for infrequently used data and the performance of mirroring for popular data. Ursa Minor brings versatility and the ability to select and change data distributions on-line to distributed cluster-based storage. To achieve AutoRAID's automatic adaptivity, Ursa Minor's versatility should be coupled with similar workload monitoring and decision-making logic.

³The physical challenges of data centers, such as heat dissipation and vibration, make storage fault tolerance less uniform across instances of a system. A deployment in an environment that struggles more with these issues will likely encounter more failures than one in a more hospitable environment.

3 Ursa Minor

Ursa Minor is a versatile cluster-based storage system. Its design and implementation grew from the desire to provide a high level of versatility in a cost-effective, cluster-based storage system.

3.1 Architecture

Ursa Minor provides storage of *objects* in the style of NASD [13] and the emerging OSD standard [31]. In general, an object consists of basic attributes (e.g., size and ACLs) and byte-addressable data. Each object has a numerical identifier (an *object ID*) in a flat name space. The system provides file-like access operations, including object CREATE and DELETE, READ and WRITE, GET_ATTRIBUTES and SET_ATTRIBUTES, etc. The primary difference from file systems is that there are no ASCII names or directories.

The main advantage of object-based storage is that it explicitly exposes more information about data stored in the system than a purely block-based storage interface like SCSI or ATA, while avoiding the specific naming and metadata semantics of any individual file system. Specifically, it exposes the set and order of data that make up each object, as well as some attributes. This information simplifies the implementation of secure direct access by clients to storage-nodes—this was the primary argument for the NASD architecture [13]. For Ursa Minor, it also facilitates the manipulation of data distribution choices for individual objects.

Like NASD and other object-based storage systems, Ursa Minor allows direct client access to storage-nodes, as illustrated in Figure 2. Clients first consult the object manager, which provides them with metadata and authorization. Afterward, they can interact directly with the storage-nodes for data operations. Metadata operations, such as object creation and deletion, are done through the object manager.

Much of Ursa Minor’s versatility is enabled by the read/write protocol family it uses for data access [15]. A *protocol family* supports different fault models in the same way that most access protocols support varied numbers of failures: by changing the number of storage-nodes accessed for reads and writes. Ursa Minor’s protocol family operates on arbitrarily-sized blocks of data. The protocol family allows each block to use any of many data encoding schemes and conform to any of many fault and timing models.

Each object’s data is stored as one or more ranges of bytes, called *slices*. Each slice is a sequence of blocks with a common block size, encoding scheme, data location, fault model, and timing model. Different slices

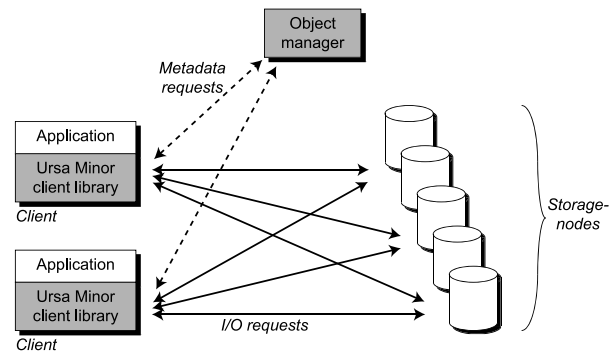


Figure 2: Ursa Minor high-level architecture. Clients use the storage system via the Ursa Minor client library. The metadata needed to access objects is retrieved from the object manager. Requests for data are then sent directly to storage-nodes.

within the same object can have different values for any of these choices. Slices allow large objects to be partitioned across multiple sets of storage-nodes. Although slices are integral to the Ursa Minor design, to simplify discussion, most of this paper refers to the data distribution of objects rather than of slices of objects.

On-line change of an object’s data distribution is arbitrated by the object manager. The data distribution can be changed for granularities as small as a single block, and clients are not prevented from accessing the object’s data during the distribution change. Such a data distribution change can alter the storage locations, encoding, fault model, timing model, or block size. Section 3.4 will describe this process in detail.

3.2 Protocol family for versatile access

Data access in Ursa Minor builds on a protocol family that supports consistent read/write access to data blocks. Each protocol family member conforms to one of two timing models, one of several fault models, and supports any threshold erasure coding scheme for data. Member implementations are distinguished by choices enacted in client-side software regarding the number of storage-nodes accessed and the logic employed during a read operation. The storage-node implementation and client-server interface is the same for all members. Pseudo-code and proofs of correctness are available in separate technical reports [16, 42].

3.2.1 Protocol family versatility

The fault tolerance provided by each member of the protocol family is determined by three independent parameters: the timing model, the storage-node failure model, and the client failure model. Each of these parameters provides tradeoffs for the performance, availability, and reliability of data.

Timing model: Protocol family members are either asynchronous or synchronous. Asynchronous members rely on no timeliness assumptions. There are no assumptions about message transmission delays or execution rates of storage-nodes. In contrast, synchronous members assume known bounds on message transmission delays between correct clients and storage-nodes as well as request processing times.

By assuming a synchronous system, storage-node crashes are detectable via timeouts. This allows clients to contact fewer storage-nodes for each operation and trust that they will get an answer from all non-faulty storage-nodes. On the other hand, if a client incorrectly “detects” that a live storage-node timed out (e.g., due to overload or a network partition), it may read inconsistent data. The asynchronous timing model is able to protect against this scenario but at the cost of additional storage-nodes and an additional round trip during writes to generate a logical timestamp.

Storage-node failure model: Each family member supports a hybrid failure model [38] for storage-nodes. Up to t storage-nodes may fail. A subset of the t failures, b , may be Byzantine faults [22], and the remaining $t - b$ must only be crash failures. Such a model can be configured across the spectrum from wholly crash (i.e., $b = 0$) to wholly Byzantine (i.e., $b = t$).

The number of storage-nodes that must be contacted for each operation increases with the number of failures that the protocol is configured to tolerate. Tolerating Byzantine failures increases the number of storage-nodes still farther. By choosing a configuration that can withstand Byzantine storage-node failures, data is protected from data corruption by storage-nodes, disk firmware errors, and buggy software.

Client failure model: Every member of the protocol family tolerates an arbitrary number of crash client failures, and some also tolerate Byzantine client failures. Client crash failures during write operations can result in subsequent read operations (by other clients) observing an incomplete write operation. As in any general storage system, an authorized client (Byzantine or otherwise) can write arbitrary values to storage. Protecting against Byzantine clients ensures only that the values written by a client are consistent (i.e., all clients reading a given version will observe the same value), not that the data itself is non-malicious. Although the implementation supports them, we do not employ the Byzantine client mechanisms in our evaluation of Ursa Minor.

3.2.2 Protocol guarantees and constraints

All members of the protocol family guarantee linearizability [17] of all correct operations. To accomplish this,

the number of storage-nodes (and thus the number of fragments, n) must conform to constraints with regard to b and t (the number of storage-node failures) as well as m (a data encoding parameter). For asynchronous members, the constraint is $2t + b + \max(m, b + 1) \leq n$. For synchronous members, the constraint is $t + \max(m, b + 1) \leq n$. Full development and proof sketches for these and other relevant constraints (e.g., read classification rules) are presented by Goodson, et al. [16].

3.2.3 Protocol operation and implementation

Each protocol family member supports read and write operations on arbitrarily-sized blocks. To write a block, the client encodes it into n fragments; any threshold-based (i.e., m -of- n) erasure code (e.g., information dispersal [29] or replication) could be used. Logical timestamps associated with each block totally order all write operations and identify fragments from the same write operation across storage-nodes. For each correct write, a client constructs a logical timestamp that is guaranteed to be unique and greater than that of the *last complete write* (the complete write with the highest timestamp). Clients form this timestamp either by issuing GET_LOGICAL_TIME requests to storage-nodes (for asynchronous members) or reading the local clock (for synchronous members). Each of the n fragments is sent to its storage-node, tagged with the logical timestamp. Storage-nodes provide fine-grained versioning, retaining a fragment version (indexed by logical timestamp) for each write request they execute.

To read a block, a client issues read requests to a subset of the n storage-nodes. From the responses, the client identifies the *candidate*, which is the fragment version returned with the greatest logical timestamp. The read operation classifies the candidate as *complete*, *incomplete* or *repairable* based on the number of read responses that share the candidate’s timestamp. If the candidate is classified as complete, then the read operation is done, and the value of the candidate is returned—by far the most common case. Only in certain cases of failures or concurrency are incomplete or repairable candidates observed. If the candidate is classified as incomplete, it is discarded, another read phase is performed to collect previous versions of fragments, and classification begins anew. This sequence may be repeated. If the candidate is repairable, it is repaired by writing fragments back to storage-nodes that do not have them (with the logical timestamp shared by the existing fragments). Then, the data is returned.

Because Byzantine storage-nodes can corrupt their data-fragments, it must be possible to detect and mask up to b storage-node integrity faults. *Cross checksums* [14] are used to detect corrupted data-fragments: a cryptographic

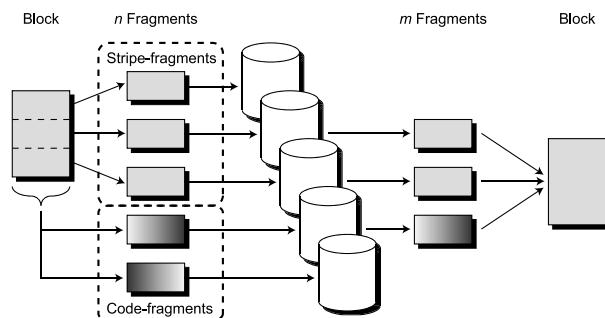


Figure 3: Erasure coding example. This example shows a 3-of-5 erasure encoding WRITE and then READ of a block. On WRITE, the original block is broken into three stripe-fragments and two code-fragments, then stored on five separate storage-nodes. On READ, any three fragments (stripe or code) can be retrieved from the storage-nodes to reconstruct the original block.

hash of each data-fragment is computed, and the set of n hashes are concatenated to form the cross checksum of the data-item.⁴ The cross checksum is stored with each data-fragment, as part of the timestamp, enabling corrupted data-fragments to be detected by clients during reads. Our implementation uses MD5 [32] for all hashes, but any collision-resistant hash could be substituted.

The protocol implementation includes a number of performance enhancements that exploit its threshold nature. For example, to improve the responsiveness of write operations, clients return as soon as the minimum number of required success responses are received; the remainder complete in the background. To make read operations more network efficient, only m read requests fetch actual fragment contents, while all fetch version histories. If necessary, after classification, extra fragments are fetched according to the candidate's timestamp.

Our implementation supports both replication and an m -of- n erasure coding scheme. If $m = 1$, then replication is used. Otherwise, our base erasure code implementation stripes the block across the first m fragments; each *stripe-fragment* is $\frac{1}{m}$ the length of the original block. Thus, concatenation of the first m fragments produces the original block. Because “decoding” with the m stripe-fragments is computationally less expensive, the implementation preferentially tries to read them. The stripe-fragments are used to generate *code-fragments* that provide the necessary redundancy (i.e., the remaining $n - m$ fragments) via Rabin’s information dispersal algorithm [29]. Figure 3 illustrates how stripe- and code-fragments are stored.

3.3 Ursa Minor components

In addition to the protocol family used for read and write operations, Ursa Minor is composed of several key com-

ponents: the storage-nodes store all data in the system; the object manager tracks system metadata and arbitrates access to objects; the client library encapsulates system interactions for applications; the NFS server allows unmodified clients to use the system.

Storage-node: Storage-nodes expose the same interface, regardless of the protocol family member being employed—read and write requests for all protocol family members are serviced identically. Clients communicate with storage-nodes via a TCP-based RPC interface. For write requests, storage-nodes provide an interface to write a fragment at a specified logical timestamp. For read requests, clients may request the version of a fragment with the greatest logical timestamp or a previous version by specifying a specific timestamp. Several other operations are supported, including retrieving the greatest logical timestamp of a fragment and retrieving a fragment’s version history.

Requests to storage-nodes address data fragments by block number because the fragment size is not fixed. Fragment sizes vary for three reasons. First, the data block size (for protocol read/write operations) is configurable and should be chosen based on data access patterns (e.g., to match the page size for database activity). Second, erasure coding results in $\frac{\text{blocksize}}{m}$ bytes per fragment. Third, the storage-node will sometimes be asked to hold information about in-progress distribution changes instead of data. On a write, the storage-node accepts whatever number of bytes the client sends and records them, indexed by the specified object ID, block number, and timestamp. On a read, the storage-node returns whatever content it holds for the specified object ID and block number.

Each write request implicitly creates a new version of the fragment, indexed by its logical timestamp. A log-structured organization [33] is used to reduce the disk I/O cost of data versioning. Multi-version b-trees [4, 36] are used by the storage-nodes to store fragments. Fragment versions are kept in a per-object b-tree indexed by a 2-tuple $\langle \text{blocknumber}, \text{timestamp} \rangle$. Like previous research [28, 37], our experiences indicate that retaining versions and performing local garbage collection come with minimal performance cost (a few percent) and that it is feasible to retain version histories for several days.

Garbage collection of old versions is used to prevent capacity exhaustion of the storage-nodes. Because write completeness is a property of a set of storage-nodes, a storage-node in isolation cannot determine which local fragment versions are safe to garbage-collect. A fragment version can be garbage-collected only if there exists a later complete write for the corresponding block. Storage-nodes classify writes by executing the read pro-

⁴In the special case of replication, a single hash is sufficient.

tol in the same manner as a client, excluding the actual data fetches. This garbage collection typically completes in the background, before writes are flushed to disk, and it can be batched across a number of data blocks.

The storage-node implementation is based on the S4 object store [36, 37]. It uses a write-back cache for fragment versions, emulating non-volatile RAM.⁵ The storage-node additionally maintains a sizeable cache of latest timestamps (including the cross checksums) associated with fragments. The hit rate of the timestamp cache is crucial for performance, as it eliminates disk accesses for storage-nodes that are queried just to ensure consistency (rather than to retrieve one of m fragments).

Object manager: The object manager maintains Ursa Minor metadata about each object, including data distribution. Clients send RPCs to the object manager to create and delete objects, access attributes, and retrieve distributions and authorizations for accessing data.

To access data, a client sends the object ID and byte offset to the object manager and, if it has appropriate access rights, gets back a *slice descriptor* and a capability. The slice descriptor details the data distribution of the slice containing the specified byte offset, including the byte range, block size, block numbers, encoding scheme, fault model, timing model, and list of storage-nodes. The object manager maintains one or more slice descriptors for each object, as needed.

The object manager implementation uses Berkeley DB [25] b-trees, stored in objects, to organize and index the Ursa Minor metadata. To enable crash recovery of the object manager, Berkeley DB was extended to support shadow paging.

The object manager implementation does not currently provide real capabilities; the field is empty and all client requests are serviced by storage-nodes without actual authorization. The “revocation” of capabilities is handled with callbacks to clients rather than communication with storage-nodes. Although not acceptable for deployment, this should not affect performance experiments.

Client library: The client library provides a byte-addressed object interface to application code, hiding the details of Ursa Minor. It includes a protocol library that, given the data distribution, handles the data encoding and protocol execution on behalf of the caller. The client library also hides other Ursa Minor details, such as interactions with the object manager. The client library is just a convenience for programmers, and it is not trusted by storage-nodes or object managers any more than application code.

⁵Our storage-nodes are battery-backed, but our implementation does not yet retain the cache contents across reboots.

NFS server: Access to data stored in Ursa Minor clearly involves non-standard protocols. To support unmodified clients, we have implemented a user-level NFS server that exports files and directories stored as objects in Ursa Minor. It supports UDP-based NFS version 3, and it uses the Ursa Minor client library to read and write data in the system. File and directory contents are stored as object data, and the NFS_ATTR structure for each is stored in the first block of the corresponding object. Directories map file names to Ursa Minor object IDs, which in turn are used as NFS file handles.

Such an NFS server is not intended as the primary method of access to a cluster-based storage system like Ursa Minor—a better choice being a parallel-access file system. However, our NFS server is convenient for incremental deployment.

3.4 On-line change of data distribution

In addition to create-time versatility, Ursa Minor supports on-line change of an object’s data distribution. This permits an administrator or automated tuning tool to correct poorly chosen distributions and to change distributions as access patterns, risks, and goals evolve.

To transition between data distributions, Ursa Minor makes use of *back-pointers*. A back-pointer is a copy of an old data distribution stored as the initial version of blocks in a new data distribution. This provides a link between the new distribution and the old, obviating the need to halt client access during the data re-encode step. A reader can follow the back-pointer to the last data written in the old distribution if no data has yet been written to the new.

A distribution change proceeds in four steps. First, the object manager installs back-pointers to the old distribution by writing them to the storage-nodes that will store the new distribution. One back-pointer is written for each new block.⁶ Second, the object manager revokes client access to the affected range of blocks. Third, the object manager updates its metadata with the new distribution and resumes issuing capabilities. Clients learn of the new distribution when they ask the object manager for access. Fourth, clients access data according to the new distribution while it is being copied, in the background, from the old to the new distribution.

During step four, clients write directly to the new distribution. When a client reads data from the new distribution, it may encounter either a back-pointer or data. If it encounters a back-pointer, the client library will proceed

⁶This operation could be batched to improve the efficiency of installing back-pointers, but back-pointer installation is not a critical-path operation.

to access the identified old distribution. Once it encounters data, it proceeds normally. Note that the data read by a client in step four may have been copied from the old distribution or it may be newly written data originating since the distribution was changed.

The Ursa Minor component that transitions data from the old distribution to the new (step four) is called a *distribution coordinator*. It copies data in the background, taking care not to write over data already written by a client to the new distribution. To ensure this behavior, the coordinator must set the timestamp for data it writes to be after the timestamp of the back-pointer but before the timestamp of any new client writes. The required gap in timestamps is created either by pausing after installing the back-pointers (in the synchronous case) or by reserving a fixed logical timestamp (in the asynchronous case).

One of the trickier aspects of data distribution change arises when the data block size is changed. Changes in the block size (used to break up the byte stream into blocks on which the protocol operates) will alter the number of blocks needed for a given byte range. This can cause conflicts between block numbers for different ranges of data bytes in an object. This problem is addressed by decoupling the block numbers used for storage from the byte offsets accessed by clients—a slice descriptor identifies the block numbers explicitly rather than having clients compute them. A new range of block numbers within the object is used for the new distribution, eliminating any conflict and enabling the use of the fixed logical timestamp (mentioned above) for the asynchronous timing model.

Ursa Minor’s approach to on-line distribution change minimizes blocking of client accesses and allows incremental application of change. Client access is only interrupted during the actual metadata update at the object manager. Further, the notion of slices allows a distribution change for a large object to be performed piecemeal rather than all at once. In addition, the coordinator can move data to the new distribution at whatever rate is appropriate. Since migration is tracked by the object manager and the distribution coordinator’s actions are idempotent, coordinators that fail can be easily restarted.

4 Evaluation

This section evaluates Ursa Minor and its versatility in three specific areas. First, it verifies that the baseline performance of NFS with Ursa Minor is reasonable. Second, it shows that Ursa Minor’s versatility provides significant benefits for different synthetic workloads. Third, it confirms that the Ursa Minor prototype can efficiently perform on-line changes of an object’s data distribution.

4.1 Experimental setup

All experiments were run using Dell PowerEdge 650 machines equipped with a single 2.66 GHz Pentium 4 processor, 1 GB of RAM, and two Seagate ST33607LW, 36 GB, 10K rpm SCSI disks. The network configuration consisted of a single Intel 82546 gigabit Ethernet adapter in each machine, connected via a Dell PowerConnect 5224 switch. The machines ran the Debian “testing” distribution and used Linux kernel version 2.4.22. The same machine type was used both as clients and storage-nodes. The storage-nodes used one of the two local disks for data; the other contained the operating system.

4.2 Baseline NFS performance

This section uses application-level benchmarks to show that Ursa Minor achieves reasonable performance. The Ursa Minor NFS server’s performance was compared to that of the Linux kernel-level NFSv3 server. Both NFS servers were configured to communicate with clients using UDP, and in both cases, they ran on dedicated machines. The Linux NFS server exported an ext3 partition that resided on a dedicated local disk. The Ursa Minor NFS server exported data stored on a single storage-node and was configured to use 384 MB of data cache and 32 MB of attribute cache. The storage-node had 640 MB of data cache and 64 MB of metadata cache.

The performance of the two systems was compared using the TPC-C and Postmark benchmarks as well as a simple source-tree compile benchmark. The TPC-C benchmark [39] simulates an on-line transaction processing database workload, where each transaction consists of a few read-modify-write operations to a small number of records. The disk locations of these records exhibit little locality. TPC-C was run on the Shore database storage manager [5] and configured to use 8 kB pages, 10 warehouses and 10 clients, giving it a 5 GB footprint. The Shore volume was a file stored on either the Linux NFS server or the Ursa Minor NFS server.

Postmark [19] is a user-level file system benchmarking tool designed to measure performance for small file workloads such as e-mail and netnews. It measures the number of transactions per second that the system is capable of supporting. A transaction is either a file create or file delete, paired with either a read or an append. The configuration parameters used were 50000 files, 20000 transactions, and 100 subdirectories. All other parameters were left as default.

We constructed the “um-build” benchmark to measure the amount of time to clean and build the Ursa Minor source tree. The benchmark copies the source tree onto a target system, then cleans and builds the Ursa Minor pro-

	Linux NFS	Ursa Minor
TPC-C	447 tpmC (2.3)	993 tpmC (13)
Postmark	17.9 tps (.01)	15.0 tps (0.0)
um-build	1069 s (5.3)	874 s (2.0)

Table 1: Macro-benchmark performance. This table shows several macro-benchmarks used to compare the performance of the Ursa Minor NFS prototype against the Linux NFS server. Standard deviations based on ten trials are listed in parentheses.

prototype. The results provide an indication of storage system performance for a programming and development workload. The source tree contained 2144 files and grew from 24 MB to 212 MB when built.

Table 1 shows performance for random I/O (TPC-C and Postmark) and system development (um-build) workloads. Overall, the two systems performed comparably in these tests. The Ursa Minor storage-node’s log-structured layout allowed it to perform better than the Linux NFS server for TPC-C and um-build. However, the extra network hop between the NFS server and storage-node added latency to I/O requests, hurting Ursa Minor’s performance for Postmark. These results show the prototype implementation is suitable for an investigation into the value of versatility.

4.3 Ursa Minor: Versatility

This section reports the results of several experiments that demonstrate the value of Ursa Minor’s versatility. These experiments access Ursa Minor directly via the client library, not through the Ursa Minor NFS server. The first three experiments explore matching distributions to workloads, and the fourth experiment shows the costs of different storage-node fault models.

For these experiments, the working set was larger than the combined client and storage-node caches. The storage-nodes used a 32 MB data cache and a 64 MB metadata cache, ensuring that most data accesses were served from disk and metadata (e.g., version history information) remained cached.

4.3.1 Specializing the data distribution

The performance and reliability of data stored in a cluster-based storage system is heavily influenced by the distribution chosen for that data. By providing versatility, a system allows data distributions to be matched to the requirements of each dataset. Without this versatility, datasets are forced to use a single distribution that is expected to perform adequately on a variety of workloads. Such compromise can lead to a significant decrease in performance, fault tolerance, or other properties.

In order to explore the trade-offs in choosing data distributions, four synthetic workloads were chosen to represent environments with different access patterns and different concerns about reliability, capacity, and performance.

Trace: This simulates trace analysis, common in research environments. It was modeled as streaming reads with a request size of 96 kB. We assumed that this data must tolerate two storage-node crash failures, since trace data can be difficult to re-acquire.

OLTP: This simulates an OLTP database workload. It was modeled as random 8 kB reads and writes in a 1:1 ratio. We assumed that this data must tolerate two storage-node crash failures, since such information is costly to lose.

Scientific: This simulates the temporary data generated during large scientific calculations. It was modeled as sequential reads and writes with a 1:1 ratio, using 96 kB requests. Because this data is generally easy to reconstruct, it did not need to tolerate any failures.

Campus: This simulates general academic computing. It was based on an analysis of the Harvard CAMPUS NFS trace [7], a mainly email workload. It was modeled as a 90% sequential and 10% random access pattern, using 8 kB requests. Fifty-five percent of accesses were reads. We assumed that this data must tolerate one storage-node crash failure.

We ran an experiment for each (workload, distribution) pair. In each experiment, six storage-nodes were used, and twelve clients ran the given workload with the specified distribution. Each client accessed a single 150 MB object.

For each workload, we determined a specialized distribution that provides it with the highest performance given the twelve client and six storage-node system configuration. We warmed the cache, then measured the throughput of the system. After trying the workload on the subset of the possible distributions where the failure requirements and block size match the workload, we chose the encoding that was most space efficient but still had throughput within 10% of optimal.

We also determined a “generic” distribution that provided good all-around performance for the four workloads. In order to determine this encoding, we ran each of the workloads on the encodings that met the failure requirements of the most stringent workload. For each encoding, we tried an 8 kB block size and all block sizes that are multiples of 16 kB up to a maximum size of 96 kB. The “generic” distribution was chosen to minimize the sum of squares degradation across the workloads. The degradation of a workload was calculated

Workload	Encoding	m	t	n	Block size
Trace	Erasure coding	2	2	4	96 kB
OLTP	Replication	1	2	3	8 kB
Scientific	Replication	1	0	1	96 kB
Campus	Replication	1	1	2	8 kB
Generic	Replication	1	2	3	8 kB

Table 2: Distributions. This table describes the data encodings for the experimental results in Figures 1 and 4. The choice for each workload was the best-performing option that met the reliability requirements and used six or fewer storage-nodes. The “generic” distribution met all workloads’ fault tolerance requirements and performed well across the set of workloads.

as the percentage difference in bandwidth between using the specialized distribution and the “generic” distribution. This penalized encodings that disproportionately hurt a specific workload. The distributions chosen for each workload, and the “generic” distribution are identified in Table 2.

Figure 1 on page 1 shows each workload using each of the five distributions in Table 2. As expected, specializing the distribution to the workload yields increased performance. The performance of a workload on a distribution specialized to another workload was poor, resulting in up to a factor of seven drop in performance. The generic distribution led to more than a factor of two drop in performance for many of the workloads. The one exception was OLTP, which performed the same with the generic encoding, since this encoding is the same as the best encoding for OLTP.

Each of the four workloads performed best when using a different data distribution. For example, the best encoding for the Trace workload was 2-of-4 erasure coding because it provided good space-efficiency as well as good performance. A 1-of-3 scheme (3-way replication) provided similar performance, but required 50% more storage space—a costly “feature” for large datasets like traces. A replicated encoding was best for OLTP because it used just one storage-node per read request (for data access). The smallest allowable amount of redundancy (i.e., the smallest t) was best, both to minimize the capacity overheads and to minimize the cost of writes.

The Scientific workload performed best with a 1-of-1 encoding because this incurred the lowest cost for writes. The best encoding for the Campus workload was a 1-of-2 scheme, which incurred the lowest number of I/Os while still providing the required fault tolerance.

4.3.2 Sharing the Ursa Minor cluster

The Ursa Minor vision is to provide a single storage infrastructure suitable for hosting many different work-

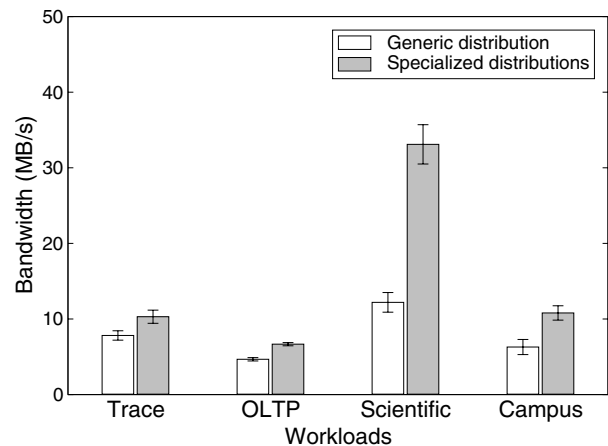


Figure 4: Matching distribution to workload on a shared cluster. This experiment shows the performance of the four workloads when they are run concurrently on a shared set of storage-nodes. The results show that, by specializing the distribution for each workload, the performance in aggregate as well as the performance for the individual workloads improves significantly. These numbers are the average of 10 trials, and the standard deviations are shown as error bars.

loads, potentially at the same time. As such, we performed experiments to determine the impact of sharing a cluster among workloads while matching the distributions to those workloads. In the previous experiment, the specialized versus generic distributions were compared in isolation. For this experiment, all workloads are run simultaneously. Figure 4 shows the performance of each workload when all four were run concurrently on the same set of storage-nodes—first with the generic distribution, then with the specialized distributions. Specializing the distribution to the workload gave improvements to all of the workloads, ranging from 32% for the Trace workload to 171% for the Scientific workload.

This shows that the cost of using a one-size-fits-all distribution is high. Moving from the generic distribution for each workload to the specialized distribution for each workload caused the aggregate throughput of the storage-nodes to increase over 96%, from 31 MB/s to 61 MB/s.

Based on Ellard’s study of Harvard’s NFS systems [7], it is apparent that real-world workloads are mixes. The studied NFS volumes showed random and sequential accesses, varied read/write ratios, and temporary as well as long-lived data. Our results show that such varied workloads could benefit greatly from the per-object versatility that Ursa Minor provides.

4.3.3 Specializing the block size

The data block size is an important factor in performance. Figure 5 shows the effect of block size on performance for two workloads in Ursa Minor. It shows

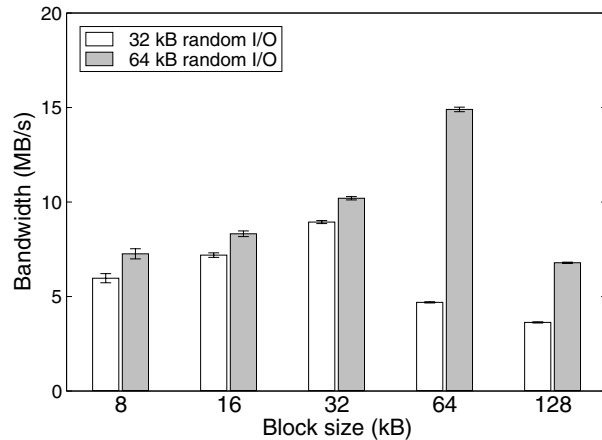


Figure 5: Matching block size to request size. This graph illustrates the importance of matching Ursa Minor’s block size to the application’s block size. In this experiment, a client performed random I/O with a 1:1 read/write ratio. The client I/O sizes were either 32 kB or 64 kB, aligned on I/O size boundaries. The block size of the object was varied between 8 kB and 128 kB. This experiment used a single client and a single storage-node. These numbers are the average of 10 trials, and the standard deviations are shown as error bars.

the bandwidth with a single client that issued an equal number of read and write requests to a single storage-node. The storage block size was varied between 8 kB and 128 kB, while the client request size remained constant. The first workload used a 32 kB request size, and the other used a 64 kB request size. Performance was best when Ursa Minor used a block size that matched the client’s requests. When the block size is smaller than the client request size, accesses have to be split into multiple requests. When the block size is too large, reads must fetch unnecessary data and writes must perform read-modify-write operations.

4.3.4 Specializing the fault model

Ursa Minor provides fault model versatility, allowing the number and types of failures tolerated to be configured on a per-object basis. Applications that can accept some risk with regard to reliability should not pay the capacity and performance costs associated with high degrees of fault tolerance. Yet, it is important to provide sufficient fault tolerance for important data.

Table 3 shows the performance of the OLTP workload when tolerating different types of faults. This experiment used 12 clients and 6 storage-nodes. The table illustrates how, in general, making the data more robust (e.g., an asynchronous timing model instead of synchronous or withstanding more failures) impacts a workload’s performance. These performance impacts would likely be unacceptable if they affected all data, but the resulting robustness benefits could be necessary for critical data.

Faults (total, byz)	Synchronous	Asynchronous
1/0	15.3 MB/s (.13)	15.8 MB/s (.15)
1/1	15.3 MB/s (.10)	11.3 MB/s (.15)
2/2	6.9 MB/s (.10)	N/A

Table 3: Fault model performance comparison. This table lists the aggregate bandwidth for the OLTP workload, using distributions that can withstand different types and numbers of storage-node failures. It shows the bandwidth as a function of the number and type of faults and the synchrony model. In all cases, replication ($m = 1$) was used. The number of storage-nodes, n , that each object was spread across, ranged from two (crash/synchronous) to five (two Byzantine/synchronous). Performance for the configuration tolerating two Byzantine failures with an asynchronous timing model is not shown since it required more than the available, six, storage-nodes. All numbers shown are the average of 10 trials, with the standard deviations shown in parentheses.

4.4 Ursa Minor: On-line change

This section describes three experiments that demonstrate Ursa Minor’s support for on-line data distribution change. To illustrate the effect of re-encoding data to match workload access characteristics and the subsequent benefits, we constructed a synthetic workload in which a single client accessed a 2 GB object randomly, using an access block size of 64 kB. In the original encoding, the object resided on a single storage-node and the block size for the data was 128 kB. During the experiment, it was re-encoded to use a 64 kB block size as well as migrated to a different storage-node.

Figure 6 illustrates the effect of re-encoding data as a function of the workload’s read:write ratio. Ursa Minor’s incremental re-encoding process is contrasted to another way of re-encoding: blocking access to the object until re-encoding completes.

Ursa Minor’s method of changing the distribution incrementally (using back-pointers) has minimal impact on the client’s requests and completes within a reasonable amount of time. This is true for both the back-pointer installation period and the coordinator copy period. Additionally, for a write-mostly workload, the role of the coordinator is less important because the workload’s writes assist the re-encoding process (back-pointers are overwritten with data as clients perform writes). A write-mostly workload also benefits quickly from the re-encoding process, because all writes are done with the new, efficient encoding.

Figure 7 illustrates the process of re-encoding for the TPC-C benchmark running over Ursa Minor’s NFS server. In this setup, the benchmark spawns 10 client threads on a single machine that accessed one warehouse with a footprint of approximately 500 MB. The database is originally encoded to use two-way replication and the block size for the database object was 64 kB. The

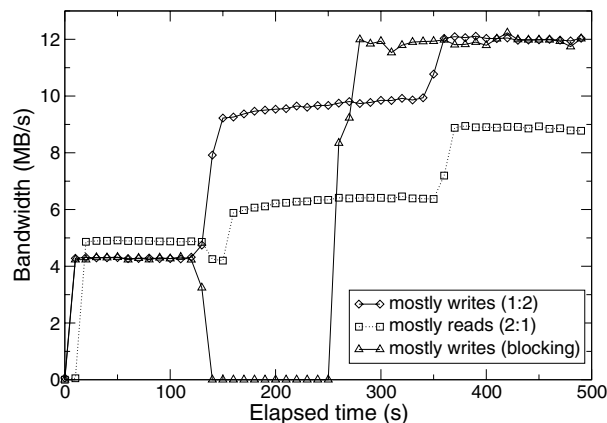


Figure 6: Distribution change. This graph shows the effect of migration and re-encoding as a function of the workload’s read/write ratio. Each point is an average of ten trials. The standard deviation for each point was less than 0.5 MB/s. The back-pointer installation began at time 130, and the migration and re-encode began at time 140. The “blocking” case completed quickly but denied access to clients during the distribution change.

database access size for TPC-C was 8 kB, causing inefficient access, especially when writing to the database. Writing an 8 kB page incurred the cost of first reading a 64 kB block and then performing a 64 kB write.

In Figure 7, the coordinator performed a re-encode in-place (using the same storage-nodes) to match the data block size to the access size. Because the re-encode used the same set of storage-nodes, there was contention between the coordinator and the client, which caused a performance drop during the back-pointer installation phase. The re-encode process took less than three minutes and upon completion, the client achieved approximately three times higher throughput from the storage-nodes.

An additional experiment was conducted that changed the distribution of the TPC-C database from a 1-of-2 (mirroring) encoding to a 4-of-5 encoding scheme. This distribution change completed in under three minutes and impacted the foreground workload by less than 5%. Such a distribution change is valuable when storage space is at a premium, because it reduces the capacity overhead from 100% to just 25%.

5 Conclusions

Versatility is an important feature for storage systems. Ursa Minor enables versatility in cluster-based storage, complementing cluster scalability properties with the ability to specialize the data distribution for each data item. Experiments show that specializing these choices

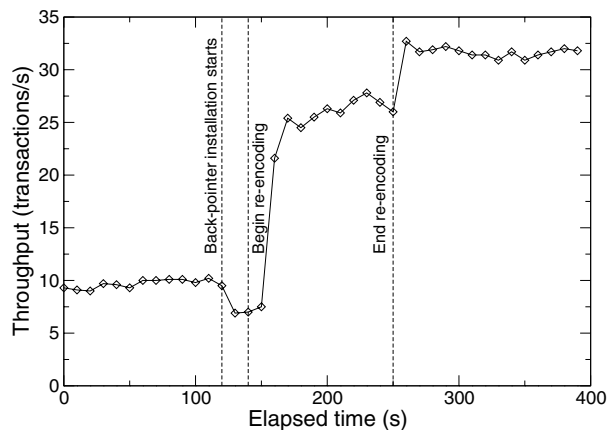


Figure 7: In-place re-encoding of a live database system. This graph shows the positive effect of re-encoding on the throughput that the TPC-C benchmark sees when accessing the underlying database. Ten trials are averaged and the standard deviation is less than 4 tps for each data point. Re-encoding changed the default block size of 64 kB to match the client’s request size of 8 kB. The database was replicated on two storage-nodes and the re-encoding happened in-place.

to access patterns and requirements can improve performance by a factor of two or more for multiple workloads. Further, the ability to change these choices on-line allows them to be adapted to observed access patterns and changes in workloads or requirements.

Acknowledgements

We thank the members and companies of the PDL Consortium (including APC, EMC, Engenio, Equallogic, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. We also thank Intel, IBM, and Seagate for hardware donations that enabled this work. This material is based on research sponsored in part by the National Science Foundation, via grant #CNS-0326453, by the Air Force Research Laboratory, under agreement number F49620-01-1-0433, by the Army Research Office, under agreement number DAAD19-02-1-0389, and by a National Science Foundation Graduate Research Fellowship. James Hendricks and Matthew Wachs are supported in part by NDSEG Fellowships, which are sponsored by the Department of Defense.

References

- [1] A. Adya, et al. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. Symposium on Operating Systems Design and Implementation. USENIX Association, 2002.

- [2] T. E. Anderson, et al. Serverless network file systems. *ACM Transactions on Computer Systems*, **14**(1):41–79. ACM, February 1996.
- [3] S. Baker and J. H. Hartman. *The Mirage NFS router*. Technical Report TR02–04. Department of Computer Science, The University of Arizona, November 2002.
- [4] B. Becker, et al. An asymptotically optimal multiversion b-tree. *VLDB Journal*, **5**(4):264–275, 1996.
- [5] M. J. Carey, et al. Shoring up persistent applications. ACM SIGMOD International Conference on Management of Data. Published as *SIGMOD Record*, **23**(2):383–394. ACM Press, 1994.
- [6] P. Corbett, et al. Row-diagonal parity for double disk failure correction. Conference on File and Storage Technologies. USENIX Association, 2004.
- [7] D. Ellard, et al. Passive NFS tracing of email and research workloads. Conference on File and Storage Technologies. USENIX Association, 2003.
- [8] EMC Corp. EMC Centera: content addressed storage system, October 2005. <http://www.emc.com/products/systems/centera.jsp?openfolder=platform>.
- [9] EqualLogic Inc. PeerStorage Overview, October 2005. http://www.equallogic.com/pages/products_technology.htm.
- [10] S. Frølund, et al. FAB: enterprise storage systems on a shoestring. Hot Topics in Operating Systems. USENIX Association, 2003.
- [11] G. R. Ganger, et al. *Self-* Storage: brick-based storage with automated administration*. Technical Report CMU-CS–03–178. Carnegie Mellon University, August 2003.
- [12] S. Ghemawat, et al. The Google file system. ACM Symposium on Operating System Principles. ACM, 2003.
- [13] G. A. Gibson, et al. A cost-effective, high-bandwidth storage architecture. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **33**(11):92–103, November 1998.
- [14] L. Gong. Securely replicating authentication services. International Conference on Distributed Computing Systems. IEEE Computer Society Press, 1989.
- [15] G. R. Goodson, et al. Efficient Byzantine-tolerant erasure-coded storage. International Conference on Dependable Systems and Networks, 2004.
- [16] G. R. Goodson, et al. *The safety and liveness properties of a protocol family for versatile survivable storage infrastructures*. Technical report CMU-PDL–03–105. Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA, March 2004.
- [17] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, **12**(3):463–492. ACM, July 1990.
- [18] IBM Almaden Research Center. Collective Intelligent Bricks, October, 2005. http://www.almaden.ibm.com/StorageSystems/autonomic_storage/CIB/index.shtml.
- [19] J. Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.
- [20] S. Kleiman. Personal communication, October 2002. Network Appliance, Inc.
- [21] A. J. Klosterman and G. R. Ganger. *Cuckoo: layered clustering for NFS*. Technical Report CMU-CS–02–183. Carnegie Mellon University, October 2002.
- [22] L. Lamport, et al. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, **4**(3):382–401. ACM, July 1982.
- [23] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. Architectural Support for Programming Languages and Operating Systems. Published as *SIGPLAN Notices*, **31**(9):84–92, 1996.
- [24] Lustre, October 2005. <http://www.lustre.org/>.
- [25] M. A. Olson, et al. Berkeley DB. Summer USENIX Technical Conference. USENIX Association, 1999.
- [26] Panasas, Inc. Panasas ActiveScale Storage Cluster, October 2005. http://www.panasas.com/products_overview.html.
- [27] D. A. Patterson, et al. A case for redundant arrays of inexpensive disks (RAID). ACM SIGMOD International Conference on Management of Data, 1988.
- [28] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. Conference on File and Storage Technologies. USENIX Association, 2002.
- [29] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, **36**(2):335–348. ACM, April 1989.
- [30] S. Rhea, et al. Pond: the OceanStore prototype. Conference on File and Storage Technologies. USENIX Association, 2003.
- [31] E. Riedel and J. Satran. OSD Technical Work Group, October 2005. http://www.snia.org/tech_activities/workgroups/osd/.
- [32] R. L. Rivest. *The MD5 message-digest algorithm*, RFC–1321. Network Working Group, IETF, April 1992.
- [33] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.
- [34] Y. Saito, et al. FAB: building distributed enterprise disk arrays from commodity components. Architectural Support for Programming Languages and Operating Systems. ACM, 2004.
- [35] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. International Conference on Distributed Computing Systems. IEEE, 1986.
- [36] C. A. N. Soules, et al. Metadata efficiency in versioning file systems. Conference on File and Storage Technologies. USENIX Association, 2003.
- [37] J. D. Strunk, et al. Self-securing storage: protecting data in compromised systems. Symposium on Operating Systems Design and Implementation. USENIX Association, 2000.
- [38] P. Thambidurai and Y. Park. Interactive consistency with multiple failure modes. Symposium on Reliable Distributed Systems. IEEE, 1988.
- [39] Transaction Processing Performance Council. TPC Benchmark C, December 2002. <http://www.tpc.org/tpcc/Revision5.1.0>.
- [40] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: a quantitative approach. International Workshop on Peer-to-Peer Systems. Springer-Verlag, 2002.
- [41] J. Wilkes, et al. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.
- [42] J. J. Wylie. *A read/write protocol family for versatile storage infrastructures*. PhD thesis. Technical report CMU-PDL–05–108, Parallel Data Laboratory, Carnegie Mellon University, October 2005.
- [43] J. J. Wylie, et al. Survivable information storage systems. *IEEE Computer*, **33**(8):61–68. IEEE, August 2000.
- [44] K. G. Yocum, et al. Anypoint: extensible transport switching on the edge. USENIX Symposium on Internet Technologies and Systems. USENIX Association, 2003.
- [45] Z. Zhang, et al. RepStore: a self-managing and self-tuning storage backend with smart bricks. International Conference on Autonomic Computing. IEEE, 2004.

Zodiac: Efficient Impact Analysis for Storage Area Networks

Aameek Singh
Georgia Institute of Technology

Madhukar Korupolu Kaladhar Voruganti
IBM Almaden Research Center

Abstract

Currently, the fields of impact analysis and policy based management are two important storage management topics that are not being treated in an integrated manner. Policy-based storage management is being adopted by most storage vendors because it lets system administrators specify high level policies and moves the complexity of enforcing these policies to the underlying management software. Similarly, proactive impact analysis is becoming an important aspect of storage management because system administrators want to assess the impact of making a change before actually making it. Impact analysis is increasingly becoming a complex task when one is dealing with a large number of devices and workloads. Adding the policy dimension to impact analysis (that is, what policies are being violated due to a particular action) makes this problem even more complex.

In this paper we describe a new framework and a set of optimization techniques that combine the fields of impact analysis and policy management. In this framework system administrators define policies for performance, interoperability, security, availability, and then proactively assess the impact of desired changes on both the system observables and policies. Additionally, the proposed optimizations help to reduce the amount of data and the number of policies that need to be evaluated. This improves the response time of impact analysis operations. Finally, we also propose a new policy classification scheme that classifies policies based on the algorithms that can be used to optimize their evaluation. Such a classification is useful in order to efficiently evaluate user-defined policies. We present an experimental study that quantitatively analyzes the framework and algorithms on real life storage area network policies. The algorithms presented in this paper can be leveraged by existing impact analysis and policy engine tools.

1 Introduction

The size and scale of the storage infrastructure of most organizations is increasing at a very rapid rate. Organizations are digitizing and persistently storing more types of

data, and are also keeping old data longer, for compliance and business intelligence mining purposes. The number of system administrators required to manage storage also increases as a function of the growth in storage because there is a limit to the amount of storage that can be managed by a single administrator. This limit is due to the number of complex tasks that a system administrator has to perform such as change analysis, provisioning, performance bottleneck analysis, capacity planning, disaster recovery planning and security analysis.

The focus of this paper is on one of these important problems namely *change analysis*. This paper provides a framework and a set of algorithms that help system administrators to proactively assess the impact of making changes in a storage area network (SAN) before making the actual change. Currently, administrators perform impact analysis manually, based on their past experience and rules of thumbs (best practices). For example, when a new host is added, the administrators have to make sure that Windows and Linux hosts are not put into the same zone or while adding a new workload, they have to ensure that the intermediate switches do not get saturated.

Manually analyzing the impact of a particular change does not scale well as the size of the SAN infrastructure increases with respect to the number of devices, best practices policies, and number of applications. Thus, deployment of new applications, hosts and storage controllers takes in the order of days or weeks because system administrators deploy the system and then reactively try to correct the problems associated with the deployment. Typically change management tools have been very reactive in their scope in that they keep snapshots of the previous state of the system, and the administrators either revert to or compare the current state with a previous state after encountering a problem.

Additionally, administrators do not have a way of assessing the impact of their proposed change with respect to a future state of the system. For example, a system administrator could potentially allocate increased band-

width to an application by taking only the current load into account. However, this could conflict with other scheduled jobs or known trends in workload surges that will increase the load on the system in the future. Thus, it is important for system administrators to assess the impact of their action not just with respect to the current state of the system but also with respect to future events.

1.1 Contributions

In order to address the above described problems, we present the **Zodiac** framework. The Zodiac framework enables system administrators to proactively assess the impact of their actions on a variety of system parameters like resource utilizations and existing system policies, before making those changes. Proactive change management analysis is an important problem and is currently receiving the deserved attention [26, 21, 23, 29]. Through Zodiac, we make the following contributions:

1. Integration with Policy based Management: The key aspect of our analysis framework is that it is tightly integrated with policy based storage management. Currently, policy-based management is being incorporated into most vendor's storage management solutions. Best-practices, service class goals, interoperability constraints, are specified as policies in the system. Thus, in essence we are combining the areas of impact analysis and policy based-management. Zodiac allows administrators to specify their rules of thumb or best practices with respect to interoperability, performance, availability, security as *policies*. It then assesses the impact of user actions by checking which of these policies are being violated or triggered. Zodiac also assesses the impact of creating new policies.

2. Scalability and Efficiency: Most system administrators want to assess the impact of their changes in real-time. A quick feedback on a proposed change encourages a system administrator to try out many alternatives. The three major components that contribute towards the execution time of impact analysis processing are: a) number of policies b) size of the storage infra-structure c) analysis time window (that is assess the impact of an action for a time window of a day, a week or a month). An impact analysis engine should be able to scale upto big SANs with 1000 hosts (found in many of today's data centers) and a few hundred policies. In this paper, we provide algorithms and data structures that help to reduce the amount of SAN data that is examined during impact analysis, the number of policies that need to be evaluated, and a framework for performing temporal impact analysis.

3. Classification Framework: One of the interesting result of the algorithm design effort in Zodiac is that we have designed a new method for classifying SAN policies based on the optimization techniques they employ. This, in turn, can also be used by general SAN policy evaluation engines to optimize their evaluation mechanisms. During policy specification period, policy designers can specify the policy type (as per this classification) as a hint to the policy engine to optimize its evaluation.

The rest of the paper is organized as follows. Section-2 provides the necessary background with respect to policy definitions, and SAN operations. Related work is presented in Section-3. The overview of our architecture is presented in Section-4 followed by the details of our implementation in Section-5. In Section-6, we discuss three important optimization algorithms that help speed up the overall analysis process. The experimental framework and the results evaluating these optimizations are presented in Section-7. We discuss related optimizations in Section-8. Finally, we conclude in Section-9.

2 Background

This section presents the necessary background material for this paper. Section-2.1 contains a discussion on the type of SAN policies considered in this paper. Section-2.2 provides the details of the storage resource models and Section-2.3 presents a list of what-if operations that one can perform. In summary, one can define various policies on the SAN resources and using our framework, analyze the impact of certain operations on both the SAN and its associated policies.

2.1 Policy Background

The term *policy* is often used by different people in different contexts to mean different things. For example, the terms *best practices*, *rule of thumbs*, *constraints*, *threshold violations*, *goals*, *rules and service classes* have been referred to as policies by different people. Currently, most standardization bodies such as IETF, DMTF, and SNIA refer to policy as a 4-field tuple where the fields correspond to an *if* condition, a *then* clause, a *priority or business value* of the policy and a *scope* that decides when the policy should be executed. The *then* clause portion can generate indications, or trigger the execution of other operation (action policies), or it can simply be informative in nature (write a message to the log). [1] describes the various SAN policies found relevant by domain experts. In this paper, within the storage area network (SAN) domain, we deal with the following types of policies:

- **Interoperability:** These policies describe what devices are interoperable (or not) with each other.
- **Performance:** These policies are violation policies that notify users if the performance of their applications (throughput, IOPs or latency) violate certain threshold values.
- **Capacity:** These policies notify users if they are crossing a percentage (threshold) of the storage space has been allotted to them.
- **Security and Access Control:** Zoning and LUN masking policies are the most common SAN access control policies. Zoning determines a set of ports that can transfer data to each other in the set. Similarly, LUN masking controls host access (via its ports) to storage volumes at the storage controller.
- **Availability:** These policies control the number of redundant paths from the host to the storage array.
- **Backup/Recovery:** These policies specify the recovery time recovery point, recovery distance, copy size and copy frequency to facilitate continuous copy and point-in-time copy solutions.

2.2 Storage Resource Model

In order to perform impact analysis, storage resource models are used to model the underlying storage infrastructure. A storage resource model consists of a schema corresponding to various **entities** like hosts, host bus adapters (HBAs), switches, controllers, the entity **attributes** (e.g. vendor, firmware level), container relationships between the entities (HBA is contained with a host), and connectivity between the entities (fabric design). These entities and attributes are used during the definition of a policy as part of the *if-condition* and the *then clause*. For a specific policy, the entities and the attributes that it uses are called its *dependent* entities and *dependent* attributes respectively. The SNIA SMI-S [28] model presents a general framework for naming and modeling storage resources.

In addition to the schema, a storage resource model also captures the behavioral aspects of the entities. The behavioral aspects, called **metrics**, represent how a resource behaves under different workload and configuration conditions. The behavioral models are either analytically specified by a domain expert [30], or deduced by observing a live system [3] or a combination of both.

Figure-1 shows the basic SAN resource model that we consider in this paper. Our resource model consists of hosts, HBAs, ports, switches, storage controllers, zones, and volume entities, and host to HBA, port to HBA, port to zone containment relationships and port to port connection relationships. In addition, there exists a *parent entity class* called *device*, which contains all the SAN

devices. The *device* entity class can be used to define global policies like *all devices should have unique WWNs*. Please note that our framework and techniques are not limited to this model only but instead can also be used in more generalized storage infrastructure models.

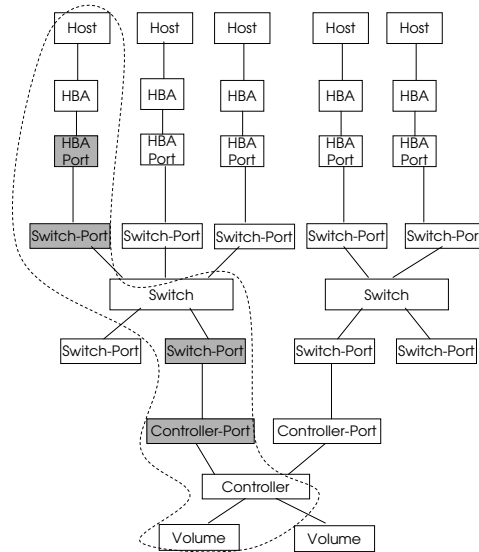


Figure 1: SAN Resource Model. A single SAN path is highlighted. Shaded ports represent zone containment.

2.3 SAN Operations:

Using Zodiac, the following types of operations can be analyzed for impact.

- *Addition/Deletion of Physical Resources* like hosts, switches, HBAs, and storage arrays.
- *Addition/Deletion of Logical Resources* like volumes and zones.
- *Access control operations* by adding or removing ports to zones or similar LUN masking operations.
- *Addition/Deletion of Workloads*. Also, the requirements (throughput, latency) of a workload can be modified. We represent a workload as a set of flows. A flow can be thought of a data path (Figure-1) between a host and a storage controller. A flow starts at a host port, and goes through intermediate switch ports and ends at a storage controller port.
- *Addition/Deletion of Policies*. Please note that we do not focus on conflict detection analysis within policies in this paper.

3 Related Work

With the growth in amount of storage resources, there has been a strong initiative for automating various manage-

ment tasks and making systems self-sufficient [2, 4, 18, 10]. Most of this research has focused on various planning tasks - capacity planning, including Minerva [2], Hippodrome [4], Ergastulum [5]; fabric planning like Appia [32, 33], and disaster recovery planning [22, 23].

Impact analysis, also referred to as “what-if” or change-management analysis, is another closely related management task. Some of the planning work described above can actually be used for such analysis. For example, Ergastulum [5] can be used to analyze storage subsystems and Keeton et al’s [23] helps in analyzing disaster recovery scenarios. Another recent work by Thereska et al. [29] provides what-if analysis using the Self-* framework [18]. There also exist tools and simulator like [21, 34] that provide impact analysis for storage controllers. Most of the what-if approaches utilize device and behavioral models for resources. Significant amount of research has been done both in developing such models [30, 3, 27, 31, 34] and using those models [14, 11, 25, 8].

Zodiac is different from existing impact analysis work, due to its close integration with policy based management. Using Zodiac, an administrator can analyze the impact of operations not only on system resources but also on system policies. In addition, the analysis accounts for all subsequent actions triggered by policy executions. As we describe later, efficient analysis of policies is non-trivial and critical for overall performance.

The Onaro SANscreen product [26] provides a similar predictive change management functionality. However, from the scarce amount of published information, we believe that they only analyze the impact for a small set of policies (mainly security) and do not consider any triggered policy actions. We believe this to be an important shortcoming, since typically administrators would specify policy actions in order to correct erroneous events and would be most interested in analyzing the impact of those triggered actions. The EMC SAN Advisor [16] tool provides support for policy evaluations, but is not an impact analysis tool. Secondly, it pre-packages its policies and does not allow specification of custom policies.

In the policies domain, there has been work in the areas of policy specification [12, 7], conflict detection [17] and resource management [24]. The SNIA-SMI [28] is also developing a policy specification model for SANs. To the best of our knowledge, there does not exist any SAN impact analysis framework for policies. [1] proposed a policy based validation framework, which is typically used as a periodic configuration checker and is not suitable for interactive impact analysis.

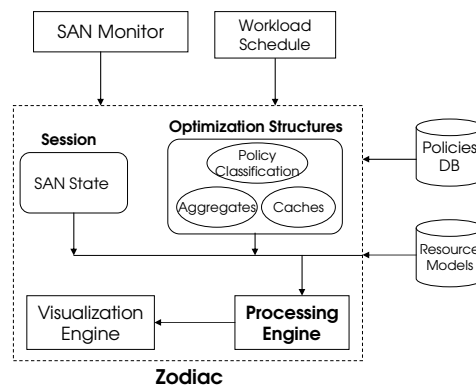


Figure 2: Architecture

4 Architecture Overview

In this section, we provide an overview of the Zodiac architecture and its interaction with other SAN modules.

4.1 Zodiac: Big Picture

The goal of the impact analysis engine, like Zodiac, is to predict the state and behavior of the SAN once a desired operation is performed. In order to evaluate the new state, the engine needs to interact with various SAN modules to get the relevant information, like device attributes, policies. The overall picture of such an eco-system is shown in Figure-2. In this eco-system, Zodiac interacts with the following modules:

- *SAN Monitor*: The foremost input requirement is the state of the SAN, which is obtained from a SAN Monitor like [15, 19, 20, 6]. It consists of the physical configuration (fabric design), resource attributes (HBA vendor, number of HBAs in a host) and logical information like Zoning/LUN-Masking.
- *Workload Schedule*: In order to predict the behavior of the SAN, Zodiac also needs to know the schedule of the workload. For example, if a backup job is scheduled for 3 AM, then the engine needs to account for the additional traffic generated due to the backup during that duration.
- *Policy Database*: A unique characteristic of the Zodiac impact analysis engine is its integration with policy based management. The policies are specified in a high level specification language like Ponder [12] or XML [1] and stored in a policy database.
- *Resource Models*: As described earlier, Zodiac uses a model based approach to evaluate the behavior of SAN resources. For this, we require a resource models database that provides such behavioral models. There has been significant work in

the area of modeling and simulation of SAN resources [30, 3, 27, 31, 11, 25, 8, 9, 34] and we leverage that. Note that the design of Zodiac is independent of the resource models and can work with any approach.

Given these modules, Zodiac takes as input the operation that the administrator wants to perform and the time at which the impact needs to be measured (immediate or after n hours) and initiates the analysis.

4.2 Zodiac: Internal Design

Internally, Zodiac engine is composed of the following primary components:

- **SAN-State:** In Zodiac, the impact analysis occurs in a *session*, during which an administrator can analyze the impact of multiple operations incrementally. So, a first operation could be - *what happens if I add two hosts?* After the engine evaluates the impact, an administrator can perform an incremental operation - *what if I add another two hosts?* The SAN state component maintains the intermediate states of the SAN, so that such incremental operations can be analyzed. When an analysis session is initialized, the SAN state is populated by the current snapshot of the SAN, obtained from the SAN Monitor.
- **Optimization Structures:** As mentioned earlier, for efficient policy evaluation, Zodiac maintains intelligent data structures that optimize the overall evaluation. These three primary structures (caches, policy classes and aggregates) are explained in detail in Section-6.
- **Processing Engine:** The processing engine is responsible for efficiently evaluating the impact of operations using the SAN state and the rest of the internal data structures. It is the main work horse of Zodiac.
- **Visualization Engine:** Another important component of Zodiac is its visualization engine. The visualization engine primarily provide two kinds of output. First, it can provide an overall picture of the SAN, with various entity metrics and can highlight interesting entities, e.g. the ones that violated certain policies. Secondly, with the incorporation of temporal analysis, an administrator can plot interesting metrics with time.

5 Zodiac: System Details

In this section, we provide the details about the internal data structures being used by Zodiac to represent SANs

(Section-5.1) and how the policy evaluation framework uses these data structures (Section-5.2). In Section-5.3, we describe the inadequacy of the current evaluation approach before proposing various optimizations in the next section.

5.1 SAN Representation

For efficient impact analysis, it is critical that SAN is represented in an optimal form. This is because all policies and resource metric computations would obtain required data through this SAN data structure. In Zodiac, the SAN is represented as a graph with entities as nodes and network links or containment relationships (HBA is contained within a host) as edges. A sample SAN as a graph is shown in Figure-1. A single SAN “path” has been highlighted. Note that it is possible to have more than one switch in the path.

Each entity in the graph has a number of attribute-value pairs, e.g. the host entity has attributes like vendor, model and OS. In addition, each entity contains pointers to its immediate neighbors (Host has a pointer to its HBA, which has a pointer to its HBA-Port and so on). This immediate neighbor maintenance and extensive use of pointers with zero duplication of data allows this graph to be maintained in memory even for huge SANs (1000 hosts).

There are two possible alternatives to this kind of immediate-neighbor representation of the SAN. We discuss the alternatives and justify our choice below:

1. **Alternative-Paths:** Assume a best practices policy requiring a Vendor-A host to be only connected to Vendor-S controller. Its evaluation would require a traversal of the graph starting at the host and going through all paths to all connected controllers. In fact many policies actually require traversals along “paths” in the graph[1]. This could indicate storing the SAN as a collection of paths and iterating over the relevant paths for each policy evaluation, preventing costly traversals over the graph. However, the number of paths in a big SAN could be enormous, and thus, prohibitive to maintain the path information in-memory. Also, the number of new paths created with an addition of a single entity (e.g. a switch) would be exponential, thus making the design unscalable.
2. **Alternative-SC:** Even without paths, it is possible to “short-circuit” the traversals by keeping information about entities further into the graph. For example, a host could also keep pointers to all connected storage. While this scheme does work for some policies, many interoperability policies, that filter paths of the graph based on some properties of

an intermediate entity, cannot be evaluated. For example, a policy that requires a Vendor-A host, *connected to a Vendor-W switch*, to be only connected to Vendor-S storage, cannot be evaluated efficiently using such a representation, since it is still required to traverse to the intermediate entity and filter based on it. However, this idea is useful and we actually use a modified version of this in our optimization schemes described later.

5.2 Policy Evaluation

In current policy evaluation engines, policies are specified in a high level specification language like Ponder [12], XML [1]. The engine converts the policy into executable code that can evaluate the policy when triggered. This uses an underlying data layer, e.g. based on SMI-S, that obtains the required data for evaluation. It is this automatic code generation, that needs to be heavily optimized for efficient impact analysis and we discuss various optimizations in Section-6.

In Zodiac, the data is obtained through our SAN data structure. For evaluating a policy like *all Vendor-A hosts should be connected to Vendor-S controllers*, a graph traversal is required (obtaining storage controllers connected to Vendor-A hosts). In order to do such traversals, each entity in the graph supports an API that is used to get to any other connected entity in the graph (by doing recursive calls to immediate neighbors). For example, hosts support a *getController()* function that returns all connected storage controllers. The functions are implemented by looking up immediate neighbors (HBAs), calling their respective *getController()* functions, aggregating the results and removing duplicates. The neighbors would recursively do the same with their immediate neighbors until the call reaches the desired entity (storage controller). Similarly for getting all connected edge switches, core switches or volumes. This API is also useful for our caching optimization. It caches the results of these function calls at all intermediate nodes for reuse in later policy evaluations.

However, even this API suffers from the limitation of the Alternative-SC scheme presented above. That is, how to obtain controllers connected only through a particular vendor switch. To facilitate this, the entity API allows for passing of filters that can be applied at intermediate nodes in the path. For our example, the filter would be *Switch.Vendor="W"*. Now, the host would call the HBA's *getController()* function with the filter *Switch.Vendor="W"*. When this call recursively reaches the switch, it would check if it satisfies the filter and only the switches that do, continue the recursion to their neighbors. Those that do not satisfy the filter return null.

The use of filters prevents unnecessary traversals on

the paths that do not yield any results (e.g. paths to the controllers connected through switches from other vendors). The filters support many comparison operations like $=$, \neq , $>$, \geq , $<$, \leq , \in and logical *OR*, *AND* and *NOT* on filters are also supported. The caching scheme incorporates filters as well (Section-6.2). The Alternative-SC presented above, can not use this filter based scheme since the possible number of filters can be enormous and thus always storing information in-memory for each such filter would be infeasible.

Notice that not all filters provide traversal optimizations. The filters that are at the "edge" of a path do not help. For example, a best practices policy - *if a Vendor-A host connected to a Vendor-W switch accesses storage from a Vendor-S controller, then the controller should have a firmware level $> x$* . In this case, the policy requires getting controllers with the filter *Switch.Vendor="W" AND Controller.Vendor="S"*. While the first term helps reduce the number of paths traversed, the second term does not – we still have to check every controller connected through the appropriate switches. Therefore, we prefer not to apply the filters at the edge, instead obtaining all edge entities (controllers in this case) and then checking for all conditions (*Vendor* and *FirmwareLevel*). This helps in bringing more useful data into the entity caches.

It is also important to mention that the traversal of the graph can also be done only for logical connections (due to zoning). This is facilitated by providing equivalent API functions for traversing links with end-points in particular zone, e.g. *getControllerLogical(Z)* obtains all connected controllers in Zone Z, i.e. all controllers reachable through a path containing ports (HBA ports, switch ports, controller ports) in zone Z.

Given the above framework, we next discuss why the current policy evaluation approach is inefficient for impact analysis.

5.3 Impact Analysis: Inadequacies of Current Approach

During impact analysis, a SAN operation can trigger multiple policies to be evaluated. For example, a host being added into the SAN would require evaluation of intrinsic host policies (policies on basic attributes of the host - *all hosts should be from a single vendor*), various host interoperability policies with other connected devices, zoning policies, and so on. With the popular policy-based autonomic computing initiative, it is highly likely that the number of policies in a SAN would be very large. So it is imperative that only the relevant set of policies are evaluated. For example, for the host-addition case, a switch and controller interoperability policy should not be evaluated.

The current policy evaluation engines [1] use a coarse classification of **scopes**. In such a scheme, each policy is designated a *Scope* to denote the class of entities, it is relevant to. In addition, it is possible to sub-scope the policy as *intra-entity* - evaluation on attributes of a single entity class or *inter-entity* - evaluation on attributes of more than one entity class [1]. The motivation for such classification is to allow administrators, to do a policy check only for a select class of entities and policies in the SAN. Unfortunately, this form of classification is not efficient for impact-analysis due to the following reasons:

- **Lack of granularity:** Consider the policy which requires a Vendor-A host to be connected only to Vendor-S storage controller. Naively, such a policy would be classified into the Host scope and the Storage scope. Thus, whenever a new host is added to the SAN, it will be evaluated and similarly, when a controller is added. However, consider the addition of a new link between an edge and a core switch. Such a link could cause hosts to be connected to new storage controllers, and thus the policy would still need to be evaluated and so, the policy also needs to be added to the Switch scope. With only scope as the classification criteria, *any* switch related event would trigger this policy. It is possible to further *sub-scope* the policy to be an *inter-entity*. However, it still will be clubbed with other switch inter-entity policies, which will cause un-necessary evaluations.
- **Failure to identify relevant SAN region:** The current scoping mechanism fails to identify the region of the SAN that needs to be traversed for policy evaluation. Consider the two policies: (a) *All Vendor-A hosts should be connected to Vendor-S storage*, and (b) *All hosts should have atleast one and atmost four disjoint paths to controllers*. Both the policies would have identical scopes (host, controller and switch) and sub-scopes (inter-entity). Now, when a switch-controller link is added to the SAN, evaluation of (a) should traverse *only* the newly created paths – ensure that all new host-storage connections satisfy the policy; there is no need to traverse a path that has already been found to satisfy that policy. However, the same is not true for (b). Its evaluation would require traversing many old paths. The current scoping mechanism fails to identify policies of type (a) and would end up evaluating many old paths in order to provide a correct and general solution.

The current policy evaluation engines also **fail to exploit the locality of data** across various policies. For example, two distinct policies might require obtaining the

storage controllers connected to the same host. In such a scenario, it is best to obtain the results for one and cache them to use it for the other. To the best of our knowledge, the current policy engines do not provide such caching schemes and rely on the underlying SMI-S data layer [1] to do this caching (which could still require evaluating expensive *join* operations). This is, in part, due to the low number of policies in current SANs and the fact that currently, the policy checking process is primarily a non-real-time, scheduled task with periodic reporting (typically daily or weekly reporting periods). As we show in Section-7, a caching scheme could have drastic performance benefits and help in interactive real-time analysis.

Such an efficiency is critical especially in the presence of **action policies**. Such policies when triggered initiate automatic operations on the SAN (“then” clause of the policy). These are typically designed as compensating actions for certain events and can do rezoning, introduce new workloads, change current workload characteristics, schedule workloads and more. For example, a policy for a write-only workload like *if Controller-A utilization increases beyond 95%, write the rest of the data on Controller-B*. Thus, when the policy is triggered, a new flow is created between the host writing the data and Controller-B, and policies related to that event need to be checked. The action might also do rezoning to put Controller-B ports in the same zone as the host and so, all zoning related policies would end up being evaluated. Overall, such a chain of events can lead to multiple executions of many policies. The caching scheme, combined with the policy classification, significantly helps in these scenarios.

6 Zodiac Impact Analysis: Optimizations

In this section, we present various optimizations in the Zodiac architecture that are critical for the scalability and efficiency of impact analysis. Zodiac uses optimizations along three dimensions.

1. **Relevant Evaluation:** Finding relevant policies and relevant regions of the SAN affected by the operation. This is accomplished using policy classification and is described in Section-6.1.
2. **Commonality of Data Accessed:** Exploiting data locality across policies or across evaluation for different entity instances. This is achieved by using caching, described in Section-6.2.
3. **Aggregation:** Efficient evaluation of certain classes of policies by keeping certain aggregate data structures. This scheme is described in Section-6.3.

All three optimizations are independent of each other and can be used individually. However, as we show later

in our results, the best performance is achieved by the combination of all three optimizations.

6.1 Policy Classification

The first policy evaluation optimization in Zodiac is policy classification. Policy classification helps in identifying the relevant regions of the SAN and the relevant policies, whenever an operation is performed. In order to identify the relevant SAN region affected by an operation, we classify the policies into four categories described below. Only the “if” condition of the policy is used for classification. Also, each policy class has a set of operations, which are the ones that can trigger the policy. This mapping of operations to policies can be made easily due to our classification scheme and is used to find the relevant set of policies.

1. **Entity-Class (EC) Policies:** These policies are defined only on the instances of a single entity class. For example, *all HBAs should be from the same vendor*, and *all Vendor-W switches must have a firmware level > x*. Such policies do not require any graph traversals, rather a scan of the list of instances of the entity class. The relevant operations for this class of policies are addition/deletion of an entity-instance or modification of a “dependent” attribute of an instance like changing the firmware level of a switch (for our second example above). Additionally, EC policies can be subdivided into two types:

- *Individual (EC-Ind) Policy:* A policy that holds on every instance of the entity class. For example, *all switches must be from Vendor-W*. This class of policies has the characteristic that whenever an instance of the entity class is added/modified, the policy only needs to be evaluated on the new member.
- *Collection (EC-Col) Policy:* A policy that holds on a collection of instances of the entity class. For example, *the number of ports of type X in the fabric is less than N* and also *all HBAs should be from the same vendor*¹. This class of policies might require checking all instances for final evaluation.

2. **Single-Path (SPTH) Policies:** These policies are defined on more than one entity on a single path of the SAN. For example, *all Vendor-A hosts must be connected to Vendor-S storage*. Importantly, SPTH policies have the characteristic that the policy is required to hold on *each* path. In our example, each and every path between hosts and storages must satisfy this policy. This characteristic implies that on application of **any** operation, there is no need to evaluate this policy on old paths. Only new paths need to be checked.

The relevant operations for these policies are addition/deletion/modification of paths or modification of a “dependent” attribute (vendor name) of a “dependent” entity (storage controller) on the path.

3. **Multiple-Paths (MPTH) Policies:** These policies are defined across multiple paths of the SAN. For example, *all hosts should have at least two and at most four disjoint paths to storage*, and *a Vendor-A host should be connected to at most five controllers*. MPTH policies cannot be decomposed to hold on individual paths for *every* operation. For the examples, adding a host requires checking only for the new paths created, whereas adding a switch-controller link requires checks on earlier paths as well. We are working on developing a notion distinguishing between the two cases². In this paper, we consider MPTH policy as affecting all paths. The relevant operations for these policies are addition/deletion/modification of paths or modification of a “dependent” attribute of a “dependent” entity on the path.

4. **Zoning/LUN-Masking (ZL) Policies:** These policies are defined on zones or LUN-Mask sets of the SAN. For example, *a zone should have at most N ports*, and *a zone should not have both windows or linux hosts*. For our discussion, we only use zone policies, though the same approach can be used for LUN-Masking policies. Notice that these policies are similar to EC policies with entity-class being analogously replaced by zones or LUN-Mask sets. Just as EC policies are defined on attributes of entity instances, ZL policies are defined on attributes of zone instances. Also similar to EC policies, Zone policies can be collection policies, requiring evaluation over multiple zones, (e.g. *the number of zones in the fabric should be at most N*)³ and individual policies, requiring evaluation only over an added/modified zone (e.g. *all hosts in the zone must be from the same vendor*). Also, within a zone, a policy might require evaluation over only the added/modified component (*Zone-Member-Ind*) or all components (*Zone-Member-Col*). An example of a *Zone-Member-Ind* policy is *all hosts in the zone should be windows*, and an example of *Zone-Member-Col* policy is *a zone should have at most N ports*. The relevant operations for this class of policies are addition/deletion of a zone instance or modification of an instance (addition/deletion of ports in the zone).

Note that the aim of this classification is not to semantically classify all conceivable policies, but rather to identify the policies that can be optimized for evaluation. Having said that, using our classification scheme, it was indeed possible to classify all policies mentioned in [1], the only public set of SAN policies collected from administrators and domain experts. The basic difference

between the classification scheme in [1] and our scheme stems from the fact that it classifies policies based on specification criteria, while we use the internal execution criteria for the classification. This helps us in generating optimized evaluation code by checking only the relevant regions of the SAN.

6.2 Caching

The second optimization we propose, uses a caching scheme to cache relevant data at *all* nodes of the SAN resource graph. Such a scheme is extremely useful in an impact-analysis framework due to the commonality of data accessed in the following scenarios:

1. *Multiple executions of a single policy:* A single policy might be executed multiple times on the same entity instance due to the chaining of actions, defined in the *then* clause of the triggered policies. Any previous evaluation data can be easily reused.

2. *Execution of a single policy for different instances of entities:* For example, consider an operation of adding a policy like *all Vendor-A hosts should be connected to Vendor-S storage*. For impact analysis, the policy needs to be evaluated for all hosts. In our immediate-neighbor scheme, for the evaluation of this policy, a host, say Host-H, would call its HBA's *getController()* function, which in turn would call its ports' *getController()* function, which would call the edge switch (say Switch-L) and so on. Now, when any other host connected to Switch-L calls its *getController()* function, it can reuse the data obtained during the previous evaluation for Host-H. Note that with no replacement, the caching implies that traversal of any edge during a policy evaluation for all entity instances is done *atmost* once. This is due to the fact that after traversing an edge $\{u,v\}$ once, the required data from v would be available in the cache at u , thus preventing its repeated traversal.

3. *Locality of data required across multiple policies:* It is also possible, and often the case, that multiple policies require accessing different attributes of the same entity. As mentioned earlier, we do not apply filters to the "edge" entities (e.g. controllers for a *getController()* call) and retrieve the full list of entities. Now, this cached entry can be used by multiple policies, even when their "dependent" attributes are different.

As mentioned earlier, the caching scheme incorporates filters as well. Whenever an API function is called with a filter, the entity saves the filter along with the results of the function call and a **cache hit** at an entity occurs only when there is a complete match, i.e. the cached entry has the same API function call as the new request and

the associated filters are also the same. This condition can be relaxed by allowing a partial match, in which the cached entry is for the same function call, but can have a more general filter. For example, assume a cache entry for *getController()* with the filter *Switch.Vendor="W"*. Now, if the new request requires controllers with the filter *Switch.Vendor="W" AND Switch.FirmwareLevel > x*, the result can be computed from the cached data itself. We leave this for future work. Also, the current caching scheme uses LRU for replacement.

6.3 Aggregation

It is also possible to improve the efficiency of policy execution by keeping certain aggregate data structures. For example, consider a policy which mandates that *the number of ports in a zone must be atleast M and atmost N*. With every addition/deletion of a port in the zone, this policy needs to be evaluated. However, each evaluation would require counting the number of ports in the zone. Imagine keeping an aggregate data structure that keeps the number of ports in every zone. Now, whenever a port is added/deleted, the policy evaluation reduces to a single check of the current count value.

We have identified the following three classes of policies that can simple aggregate data structures:

1. *Unique:* This class of policies require a certain attribute of entities to be unique. For example, policies like *the WWNs of all devices should be unique, all Fibre Channel switches must have unique domain IDs*. For these class of policies, a hashtable is generated on the attribute and whenever an operation triggers this policy, the policy is evaluated by looking up that hashtable. This aggregate data structure can provide good performance improvements especially in big SANs (Section-7). Note that such an aggregate is kept only for EC and ZL policies (where it is easier to identify addition/deletion). However, there does not appear to be any realistic SPTH or MPTH unique policies.

2. *Counts:* These policies require counting a certain attribute of an entity. Keeping the count of the attribute prevents repeated counting whenever the policy is required to be evaluated. Instead, the count aggregate is incremented/decremented when the entity is added/deleted. A count aggregate is used only for EC and ZL policies. While SPTH and MPTH count policies do exist (e.g. *there must be atmost N hops between host and storage* and *there must be atleast one and atmost four disjoint paths between host and storage* respectively), maintaining the counts is tricky and we do not use an aggregate.

3. *Transformable*: It is easy to see that the policy evaluation complexity is roughly of the order $EC-Ind \approx Zone-Member-Ind < EC-Col \approx Zone-Member-Col < SPTH < MPTH$. It is actually possible to transform many policies into a lower complexity policy by keeping additional information about some of the dependent entities. For example, consider a policy like *all storage should be from the same vendor*. This policy is an EC-Col for entity class - Storage. However, keeping information about the current type of storage (T) in the system, the policy can be reduced to an equivalent EC-Ind policy – *all storage should be of type T* . Similarly, a Zone-Member-Col policy like *a zone should not be both windows and linux hosts* can be transformed into multiple Zone-Member-Ind policies *there should be only type T_i hosts in zone Z_i* , where T_i is the current type of hosts in the Z_i . For these transformed policies, a pointer to the entity that provides the value to aggregate is also stored. This is required, since when the entity is deleted, the aggregate structure can be invalidated (can be re-populated using another entity, if existing).

For all other policies, we currently do not use any aggregate data structures.

7 Experimental Setup and Results

In this section, we evaluate our proposed optimizations as compared to the base policy evaluation provided by current engines. We start by describing our experimental setup beginning with the policy set.

7.1 Microbenchmarks

With the policy based management being in a nascent state so far, there does not exist any public set of policies that is used in a **real** SAN environment. The list of policies contained in [1] is indicative of the *type* of possible policies and not an accurate “trace” for an actual SAN policy set. As a result, it is tough to analyze the overall and cumulative benefits of the optimizations for a real SAN. To overcome this, we try to demonstrate the benefits of optimizations for different categories of policies individually. As mentioned earlier, since we have been able to classify all policies in [1] according to our scheme, the benefits would be additive and overall useful for a real SAN as well. In addition, this provides a good way of comparing the optimization techniques for each policy class.

We selected a set of 7 policies (Figure-3) as our working sample. Four of them are EC policies, two are path policies (SPTH and MPTH) and one is a zone policy. All 7 policies are classified according to the classification mechanisms presented in Section-6.1. Any aggregates that are possible for policies are also shown. For

#	Policy	Classification
1	Every HBA that has a vendor name V and model M should have a firmware level either n1, n2 or n3	EC
2	No two devices in the system can have the same WWN (World Wide Name)	EC Unique
3	The number of ports of type X in the fabric is less than N	EC Counts
4	The SAN should not have mixed storage type such as SSA, FC and SCSI parallel	EC-Col to EC-Ind (Transform)
5	An ESS array is not available to open systems if an iSeries system is configured to array	SPTH
6	A HBA cannot be used to access both tape and disk drives	MPH to SPTH (Transform)
7	No two different host types should exist in the same zone	Zone-Member-Col to Zone-Member-Ind

Figure 3: Policy Set

this set of policies, we will evaluate the effectiveness of our optimizations individually.

7.2 Storage Area Network

An important design goal for Zodiac was scalability and ability to perform impact analysis efficiently even on huge SANs of 1000 hosts and 200 controllers. Since it was not possible to construct such large SANs in the lab, for our experiments, we emulated four different sized SANs. Please note that in practice, Zodiac can work with any real SAN using an SMI-S compliant data store.

In our experimental SANs, we used hosts with two HBAs each and each HBA having two ports. Storage controllers had four ports each. The fabric was a core-edge design with 16-port edge and 128-port core switches. Each switch left certain ports unallocated, to simulate SANs constructed with future growth in mind. The four different configurations correspond to different number of hosts and controllers:

- **1000-200**: First configuration is an example of a big SAN, found in many data centers. It consists of 1000 hosts and 200 controllers with each host accessing all controllers (full connectivity). There were 100 zones.
- **750-150**: This configuration uses 750 hosts and 150 controllers with full connectivity. There were 75 zones.
- **500-100**: This configuration has 500 hosts, 100 controllers and 50 zones.
- **250-50**: This configuration is a relatively smaller SAN with 250 hosts, 50 controllers and 25 zones.

7.3 Implementation Techniques

For our experiments, we evaluate each of the policies in the policy-set with the following techniques:

- **base**: This technique is the naive implementation, in which there is no identification of the relevant region of the SAN. Only information available is the vanilla scope of the policy. Due to lack of classification logic, this implementation implies that the evaluation engine uses the same logic of code generation for all policies (check for all paths and all entities). Also, there is no intermediate caching and no aggregate data structures are used.
- **class**: This implementation uses the classification mechanism on top of the base framework. Thus, it is possible to optimize policies by only evaluating over a relevant SAN region, but no caching or aggregation is used.
- **cach**: This implementation technique caching on top of the *base* framework. No classification or aggregation is used.
- **agg**: This implementation technique only uses aggregate data structures for the policies (where ever possible). There is no caching or classification.
- **all**: This implementation uses a combination of all three optimization techniques.

Using these five classes of implementation, we intend to show (a) inadequacy of the base policy, (b) advantages of each optimization technique and (c) the performance of the **all** implementation. Zodiac is currently running on a P4 1.8 GHz machine with 512 MB RAM.

7.4 Policy Evaluation

In this section, we present our results of evaluating each policy 100 times to simulate scenarios of chaining and execution for multiple instances (e.g. adding 10 hosts). The policies are evaluated for all four SAN configurations (X-axis). The Y-axis plots the time taken to evaluate the policies in milliseconds. The results have been averaged over 10 runs.

7.4.1 Policy-1

“Every HBA that has a vendor name *V* and model *M* should have a firmware level either *n1*, *n2* or *n3*”

The first policy requires a certain condition to hold on an HBA entity class. We analyze the impact of the policy when an HBA is added. The *base* implementation will trigger the HBA scope and try to evaluate this policy. Due to its lack of classification logic, it will end up evaluating the policy afresh and thus, for all HBA instances. The *class* implementation would identify it to be an EC-Ind policy and only evaluate on the new HBA entity. The *cach* implementation does not help since there is no traversal of the graph. The *agg* implementation also does not help. As a result, *all* implementation

is equivalent to having only *class* optimization. Figure-4 shows the results for the different SAN configurations.

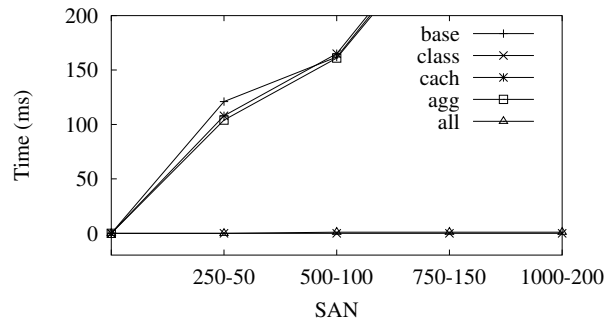


Figure 4: Policy-1. *class*, *all* provide maximum benefit

As seen from the graph, there is a significant difference between the best optimized evaluation (*all*) and the *base* evaluation. Also, as the size of the SAN increases, the costs for the *base* implementation increase, while the *all* implementation stays the same, since irrespective of SAN size, it only needs to evaluate the policy for the newly added HBA.

7.4.2 Policy-2

“No two devices in the system can have the same WWN.”

The second policy ensures uniqueness of world wide names (WWNs). We analyze the impact when a new host is added. The *base* implementation will trigger the *device* scope without classification logic and check that all devices have unique WWNs. The *class* implementation will only check that the new host has a unique WWN. The *cach* implementation performs similar to *base*. The *agg* implementation will create a hashtable, and do hashtable lookups. The *all* implementation also uses the hashtable and only checks the new host.

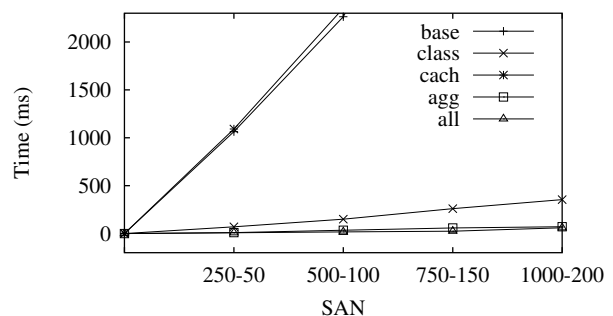


Figure 5: Policy-2. *agg*, *all* provide maximum benefit

As Figure-5 shows, *agg* and *all* perform much better than the *base* implementation. *class* performs better than *base* by recognizing that only the new host needs to be checked.

7.4.3 Policy-3

“The number of ports of type X in the fabric is less than N.”

The third policy limits the total number of ports in the fabric. We analyze the impact of adding a new host with 4 ports to the SAN. For each added port, the *base* implementation will count the total number of ports in the fabric. The *class* implementation performs no better, since it is an EC-Col policy. The *cach* implementation also does not help. The *agg* implementation keeps a count of the number of ports and only increments the count and checks against the upper limit. The *all* implementation also exploits the aggregate keeping.

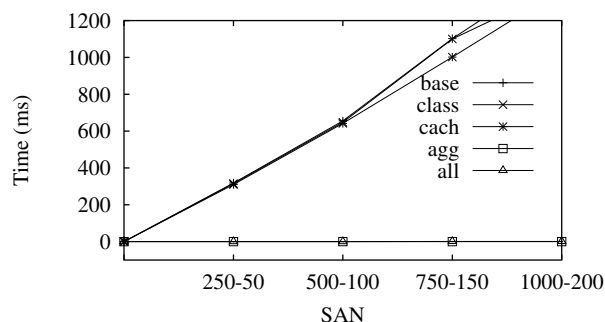


Figure 6: Policy-3. *agg*, *all* provide maximum benefit

As can be seen from Figure-6, *agg* and *all* perform significantly better due to the ability of aggregating the required information for the policy evaluation.

7.4.4 Policy-4

“The SAN should not have mixed storage type such as SSA, FC and SCSI parallel”

The fourth policy ensures that the SAN has uniform storage type. For this policy, we analyze the impact of adding a new storage controller. The *base* implementation will trigger the storage scope and evaluate the policy ensuring all controllers are the same type. The *cach* implementation will not help. The *class* implementation only checks that the newly added controller is the same type as every other controller. The *agg* implementation will transform the policy to an EC-Ind policy by keeping an aggregate value of the current controller type, *T* in the SAN. However, without classification, it would end up checking that all controllers have the type *T*. The *all* implementation combines the classification logic and the aggregate transformation to only check for the new controller.

Figure-7 shows the result with *all* performing the best, while *class* and *agg* doing better than *base* and *cach*. The difference between the best and poor implementations is small since the total number of controllers is small.

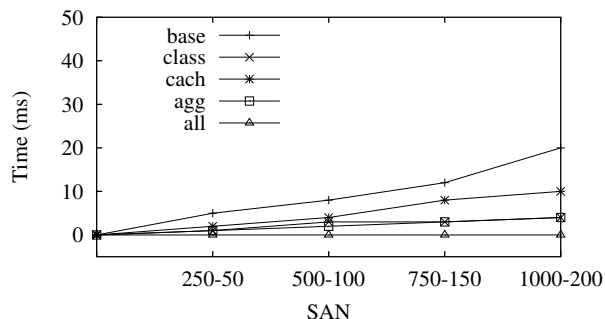


Figure 7: Policy-4. *all* provides maximum benefit

7.4.5 Policy-5

“An ESS array is not available to open systems if an iSeries system is configured to array.”

The fifth policy is an SPTH policy that checks that an iSeries open systems host does not work if an ESS array is used with the controller. We analyze the impact of adding a new host to the SAN for this policy. The *base* implementation ensures that all iSeries open systems hosts do not have any ESS controllers connected to them. This requires calling the *getController()* API functions of the host entities and will cause traversals of the graph for all host-storage connections. The *class* implementation identifies that it being an SPTH, only the new created paths (paths between the newly added host and the connected storage controllers) need to be checked. The *cach* implementation will run similar to *base*, but will cache all function call results at intermediate nodes (As mentioned before, it would mean that each edge will be traversed at most once). The *agg* implementation does not help and the *all* implementation would use both the classification logic and the caching.

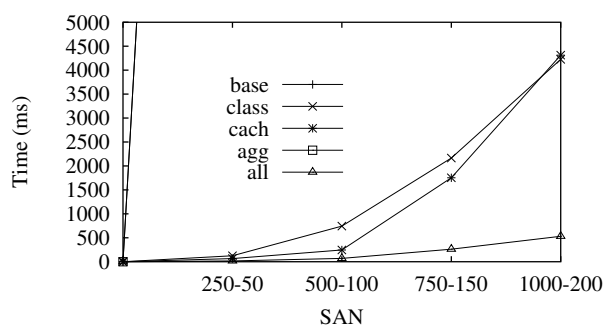


Figure 8: Policy-5. Only *all* provides maximum benefit

As shown in Figure-8, the *base* and *agg* implementation perform extremely poorly (multiple orders of magnitude in difference) due to multiple traversals for the huge SAN graph. On the other hand, *class* and *cach* are able

to optimize significantly and their combination in the *all* implementation provides drastic overall benefits. It also scales extremely well with the increasing SAN size.

7.4.6 Policy-6

“A HBA cannot be used to access both tape and disk drives.”

The sixth policy is an MPTH policy which requires checking that each HBA is either connected to tape or disk storage. We analyze the impact of adding a host with 2 HBAs to the SAN. The *base* implementation would check the policy for all existing HBAs. The *cach* implementation would do the same, except the caching of results at intermediate nodes. The *class* implementation does not optimize in this case since it considers it an MPTH policy and checks for all paths (Section-6.1). The *agg* implementation transforms the policy to an SPTH by keeping aggregate for the type of storage being accessed, but checks for all HBAs due to the lack of classification logic. The *all* implementation is able to transform the policy and then use the SPTH classification logic to only check for the newly added HBAs.

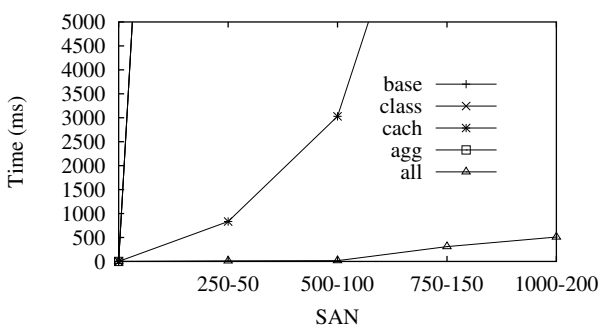


Figure 9: Policy-6. Only *all* provides maximum benefit

Figure-9 shows the results. The *base*, *class* and *agg* implementation perform much poorly then the *cach* implementation, since the *cach* implementation reuses data collected once for the other. The *all* implementation performs the best by combining all optimizations.

7.4.7 Policy-7

“No two different host types should exist in the same zone.”

The seventh policy requires that all host types should be the same in zones. We analyze the impact of adding a host HBA port to a zone. The *base* implementation would check that all hosts in each zone are of the same type. The *class* implementation would check only for the affected zone. The *cach* implementation would be the same as *base*. The *agg* implementation would keep an aggregate host type for each zone and check the policy for all zones. The *all* implementation would combine the

aggregate with the classification and only check for the affected zone, that the new host has the same type as the aggregate host type value. Figure-10 shows the results. Again *all* implementation performs the best, though the difference between all implementations is small.

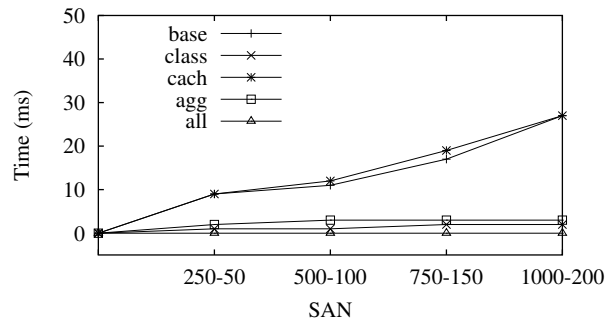


Figure 10: Policy-7. *all* provides maximum benefit

8 Discussion

One of the main objectives of the Zodiac framework is to efficiently perform impact analysis for policy enabled SANs. It is important to note that the optimizations described in this paper attack the problem at a higher conceptual level, manipulating the design and evaluation of policies. In the overall impact analysis picture, more optimizations will be plugged-in at other layers. For example, another layer of optimizations is while obtaining the data required for policy evaluation from the SMI-S data provider. In the CIM architecture [13], the CIM client obtains data from the provider over the network. This process can be optimized by techniques like batching of requests, pre-fetching and data caching at the client. Another important layer is the query language used for evaluating the policies. For example, it is possible to evaluate the policies using SQL by designing a local database scheme which is populated by the CIM client. While we continue to investigate such optimizations, Zodiac has been designed in a manner that it is easily possible to accommodate these into the overall framework.

9 Conclusions and Future Work

In this paper, we presented Zodiac - an efficient impact analysis framework for storage area networks. Zodiac enables system administrators to do proactive change analysis, by evaluating the impact of their proposed changes. This analysis is fast, scalable and thorough. It includes the impact on SAN resources, existing policies and also, due to the actions triggered by any of the violated policies. In order to make the system efficient,

we proposed three optimizations - classification, caching and aggregation. Based on our analysis and experimental results, we find that each optimization has a niche of evaluation scenarios where it is most effective. For example, caching helps the most during the evaluation of path policies. Overall, a combination of the three optimization techniques yields the maximum benefits.

In future, we intend to follow two lines of work. The first includes developing more optimization techniques - smarter analysis for MPTH policies and use of parallelism (works for SPTH policies), to name a few and the design of a policy specification language that allows determination of these optimizations. The second direction explores the integration of the impact analysis framework with various SAN planning tools in order to provide better overall designs and potentially suggesting appropriate system policies for given design requirements.

References

- [1] AGRAWAL, D., GILES, J., LEE, K., VORUGANTI, K., AND ADIB, K. Policy-Based Validation of SAN Configuration. *POLICY '04*.
- [2] ALVAREZ, G., BOROWSKY, E., GO, S., ROMER, T., SZENDY, R., GOLDING, R., MERCHANT, A., SPASOJEVIC, M., VEITCH, A., AND WILKES, J. Minerva: An Automated Resource Provisioning tool for large-scale Storage Systems. *ACM Trans. Comput. Syst.* 19, 4 (2001).
- [3] ANDERSON, E. Simple table-based modeling of storage devices. *HP Labs Tech Report HPL-SSP-2001-4* (2001).
- [4] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: Running Circles Around Storage Administration. In *FAST* (2002).
- [5] ANDERSON, E., KALLAHALLA, M., SPENCE, S., SWAMINATHAN, R., AND WANG, Q. Ergastulum: Quickly finding near-optimal Storage System Designs. *HP Tech Report HPL-SSP-2001-5* (2001).
- [6] ASSOCIATES, C. BrightStor. <http://www.ca.com> (2005).
- [7] BANDARA, A., LUPU, E., AND RUSSO, A. Using Event Calculus to Formalise Policy Specification and Analysis. *POLICY '03*.
- [8] BERENBRINK, P., BRINKMANN, A., AND SCHEIDELER, C. SimLab - A Simulation Environment for Storage Area Networks. In *Workshop on Parallel and Distributed Processing (PDP)* (2001).
- [9] BUCY, J., GANGER, G., AND CONTRIBUTORS. The DiskSim Simulation Environment. *CMU-CS-03-102* (2003).
- [10] CHAUDHURI, S., AND NARASAYYA, V. AutoAdmin 'what-if' Index Analysis Utility. In *SIGMOD* (1998).
- [11] COHEN, I., CHASE, J., GOLDSZMIDT, M., KELLY, T., AND SYMONS, J. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI* (2004).
- [12] DAMIANOU, N., DULAY, N., LUPU, E., AND SLOMAN, M. The Ponder Policy Specification Language. In *POLICY* (2001).
- [13] DMTF. Common Information Model. <http://www.dmtf.org>.
- [14] DOYLE, R., CHASE, J., ASAD, O., W., AND VAHDAT, A. Model-based resource provisioning in a web service utility. In *USITS* (2003).
- [15] EMC. Control Center. <http://www.emc.com> (2005).
- [16] EMC. SAN Advisor. <http://www.emc.com> (2005).
- [17] FU, Z., WU, S., HUANG, H., LOH, K., GONG, F., BALDINE, I., AND XU, C. IPSec/VPN Security Policy: Correctness, Conflict Detection, and Resolution. In *Workshop on Policies for Distributed Systems and Networks* (2001).
- [18] GANGER, G., STRUNK, J., AND KLOSTERMAN, A. Self-* Storage: Brick-based Storage with Automated Administration. *CMU Tech Report CMU-CS-03-178* (2003).
- [19] HP. StorageWorks SAN.
- [20] IBM. TotalStorage Productivity Center.
- [21] INTELLIMAGIC. Disc Magic. <http://www.intellimagic.nl> (2005).
- [22] KEETON, K. Designing for disasters. In *FAST* (2004).
- [23] KEETON, K., AND MERCHANT, A. A Framework for Evaluating Storage System Dependability. In *International Conference on Dependable Systems and Networks (DSN'04)* (2004).
- [24] LYMBERPOULOS, L., LUPU, E., AND SLOMAN, M. An Adaptive Policy-based Framework for Network Services Management. *Journal of Networks and System Management* 11, 3 (2003).
- [25] MOLERO, X., SILLA, F., SANTONJA, V., AND DUATO, J. Modeling and Simulation of Storage Area Networks. In *MASCOTS* (2000).
- [26] ONARO. SANSscreen. <http://www.onaro.com>.
- [27] RUEMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer* 27, 3 (1994).
- [28] SNIA. Storage Management Initiative. <http://www.snia.org>.
- [29] THERESKA, E., NARAYANAN, D., AND GANGER, G. Towards self-predicting systems: What if you could ask "what-if"? In *Workshop on Self-adaptive and Autonomic Comp. Systems* (2005).
- [30] VARKI, E., MERCHANT, A., XU, J., AND QIU, X. Issues and Challenges in the Performance Analysis of Real Disk Arrays. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004).
- [31] WANG, M., AU, K., AILAMAKI, A., BROCKWELL, A., FALOUTSOS, C., AND GANGER, G. Storage device performance prediction with CART models. *SIGMETRICS Performance Eval. Review* 32, 1 (2004).
- [32] WARD, J., SULLIVAN, M., SHAHOUMIAN, T., AND WILKES, J. Appia: Automatic Storage Area Network Fabric Design. In *FAST '02*.
- [33] WARD, J., SULLIVAN, M., SHAHOUMIAN, T., WILKES, J., WU, R., AND BEYER, D. Appia and the HP SAN Designer: Automatic Storage Area Network Fabric Design. In *HP Tech. Conference* (2003).
- [34] WILKES, J. The Pantheon Storage-System Simulator. *HP Labs Tech Report HPL-SSP-95-14* (1995).

Notes

¹This policy is also a collection policy since in order to evaluate the policy for the new instance, it is required to get information about existing instances.

²Informally, typically an operation affecting only the "principal" entity of the policy (host in the examples) does not require checking old paths.

³Such a policy is required since the switches have a limit on the number of zones they can handle

Journal-guided Resynchronization for Software RAID

Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin, Madison

Abstract

We investigate the problem of slow, scan-based, software RAID resynchronization that restores consistency after a system crash. Instead of augmenting the RAID layer to quicken the process, we leverage the functionality present in a journaling file system. We analyze Linux ext3 and introduce a new mode of operation, declared mode, that guarantees to provide a record of all outstanding writes in case of a crash. To utilize this information, we augment the software RAID interface with a verify read request, which repairs the redundant information for a block. The combination of these features allows us to provide fast, journal-guided resynchronization. We evaluate the effect of journal-guided resynchronization and find that it provides improved software RAID reliability and availability after a crash, while suffering little performance loss during normal operation.

1 Introduction

Providing reliability at the storage level often entails the use of RAID [8] to prevent data loss in the case of a disk failure. High-end storage arrays use specialized hardware to provide the utmost in performance and reliability [6]. Unfortunately, these solutions come with multi-million dollar price tags, and are therefore infeasible for many small to medium businesses and organizations.

Cost-conscious users must thus turn to commodity systems and a collection of disks to house their data. A popular, low-cost solution for reliability in this arena is software RAID [15], which is available on a range of platforms, including Linux, Solaris, FreeBSD, and Windows-based systems. This software-based approach is also attractive for specialized cluster-in-a-box systems. For instance, the EMC Centra [5] storage system is built from a cluster of commodity machines, each of which uses Linux software RAID to manage its disks.

Unfortunately, in life as in storage arrays, you get what you pay for. In the case of software RAID, the

lack of non-volatile memory introduces a *consistent update* problem. Specifically, when a write is issued to the RAID layer, two (or more) disks must be updated in a consistent manner; the possibility of crashes makes this a challenge. For example, in a RAID-5 array, if an untimely crash occurs after the parity write completes but before the data block is written (*i.e.*, the two writes were issued in parallel but only one completed), the stripe is left in an inconsistent state. This inconsistency introduces a *window of vulnerability* – if a data disk fails before the stripe is made consistent, the data on that disk will be lost. Automatic reconstruction of the missing data block, based on the inconsistent parity, will silently return bad data to the client.

Hardware RAID circumvents this problem gracefully with non-volatile memory. By buffering an update in NVRAM until the disks have been consistently updated, a hardware-based approach avoids the window of vulnerability entirely. The outcome is ideal: both performance and reliability are excellent.

With current software-based RAID approaches, however, a performance/reliability trade-off must be made. Most current software RAID implementations choose performance over reliability [15]: they simply issue writes to the disks in parallel, hoping that an untimely crash does not occur in between. If a crash does occur, these systems employ an expensive *resynchronization* process: by scanning the entire volume, such discrepancies can be found and repaired. For large volumes, this process can take hours or even days.

The alternate software RAID approach chooses reliability over performance [3]. By applying write-ahead logging within the array to record the location of pending updates before they are issued, these systems avoid time-consuming resynchronization: during recovery, the RAID simply repairs the locations as recorded in its log. Unfortunately, removing the window of vulnerability comes with a high performance cost: each update within the RAID must now be preceded by a syn-

chronous write to the log, greatly increasing the total I/O load on the disks.

To solve the consistent update problem within software RAID, and to develop a solution with both high performance and reliability, we take a global view of the storage stack: how can we leverage functionality within other layers of the system to assist us? In many cases, the client of the software RAID system will be a modern journaling file system, such as the default Linux file system, ext3 [16, 17, 18], or ReiserFS [9], JFS [1], or Windows NTFS [12]. Although standard journaling techniques maintain the consistency of file system data structures, they do not solve the consistent update problem at the RAID level. We find, however, that journaling can be readily augmented to do so.

Specifically, we introduce a new mode of operation within Linux ext3: *declared mode*. Before writing to any permanent locations, declared mode records its intentions in the file system journal. This functionality guarantees a record of all outstanding writes in the event of a crash. By consulting this activity record, the file system knows which blocks were in the midst of being updated and hence can dramatically reduce the window of vulnerability following a crash.

To complete the process, the file system must be able to communicate its information about possible vulnerabilities to the RAID layer below. For this purpose, we add a new interface to the software RAID layer: the *verify read*. Upon receiving a verify read request, the RAID layer reads the requested block as well as its mirror or parity group and verifies the redundant information. If an irregularity is found, the RAID layer re-writes the mirror or parity to produce a consistent state.

We combine these features to integrate journal-guided resynchronization into the file system recovery process. Using our record of write activity vastly decreases the time needed for resynchronization, in some cases from a period of days to mere seconds. Hence, our approach avoids the performance/reliability trade-off found in software RAID systems: performance remains high and the window of vulnerability is greatly reduced.

In general, we believe the key to our solution is its *cooperative* nature. By removing the strict isolation between the file system above and the software RAID layer below, these two subsystems can work *together* to solve the consistent update problem without sacrificing either performance or reliability.

The rest of the paper is organized as follows. Section 2 illustrates the software RAID consistent update problem and quantifies the likelihood that a crash will lead to data vulnerability. Section 3 provides an introduction to the ext3 file system and its operation. In Section 4, we analyze ext3's write activity, introduce ext3 declared mode and an addition to the software RAID interface,

and merge RAID resynchronization into the journal recovery process. Section 5 evaluates the performance of declared mode and the effectiveness of journal-guided resynchronization. We discuss related work in Section 6, and conclude in Section 7.

2 The Consistent Update Problem

2.1 Introduction

The task of a RAID is to maintain an invariant between the data and the redundant information it stores. These invariants provide the ability to recover data in the case of a disk failure. For RAID-1, this means that each mirrored block contains the same data. For parity schemes, such as RAID-5, this means that the parity block for each stripe stores the exclusive-or of its associated data blocks.

However, because the blocks reside on more than one disk, updates cannot be applied atomically. Hence, maintaining these invariants in the face of failure is challenging. If a crash occurs during a write to an array, its blocks may be left in an inconsistent state. Perhaps only one mirror was successfully written to disk, or a data block may have been written without its parity update.

We note here that the consistent update problem and its solutions are distinct from the traditional problem of RAID disk failures. When such a failure occurs, all of the redundant information in the array is lost, and thus all of the data is vulnerable to a second disk failure. This situation is solved by the process of reconstruction, which regenerates all of the data located on the failed disk.

2.2 Failure Models

We illustrate the consistent update problem with the example shown in Figure 1. The diagram depicts the state of a single stripe of blocks from a four disk RAID-5 array as time progresses from left to right. The software RAID layer residing on the machine is servicing a write to data block **Z**, and it must also update the parity block, **P**. The machine issues the data block write at time 1, it is written to disk at time 3, and the machine is notified of its completion at time 4. Similarly, the parity block is issued at time 2, written at time 5, and its notification arrives at time 6. After the data write to block **Z** at time 3, the stripe enters a *window of vulnerability*, denoted by the shaded blocks. During this time, the failure of any of the first three disks will result in data loss. Because the stripe's data and parity blocks exist in an inconsistent state, the data residing on a failed disk cannot be reconstructed. This inconsistency is corrected at time 5 by the write to **P**.

We consider two failure models to allow for the possibility of independent failures between the host machine

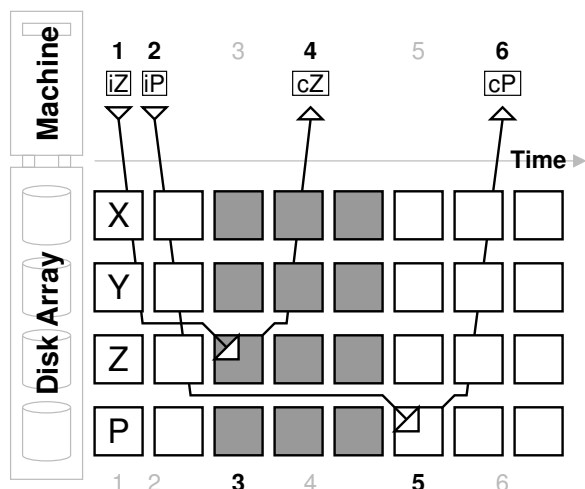


Figure 1: **Failure Scenarios.** The diagram illustrates the sequence of events for a data block write and a parity update to a four disk RAID-5 array as time progresses from left to right. The boxes labeled *i* indicate a request being issued, and those labeled *c* represent completions. The shaded blocks denote a window of vulnerability.

and the array of disks. We will discuss each in turn and relate their consequences to the example in Figure 1. The *machine failure model* includes events such as operating system crashes and machine power losses. In our example, if the machine crashes between times 1 and 2, and the array remains active, the stripe will be left in an inconsistent state after the write completes at time 3.

Our second model, the *disk failure model*, considers power losses at the disk array. If such a failure occurs between time 3 and time 5 in our example, the stripe will be left in a vulnerable state. Note that the disk failure model encompasses non-independent failures such as a simultaneous power loss to the machine and the disks.

2.3 Measuring Vulnerability

To determine how often a crash or failure could leave an array in an inconsistent state, we instrument the Linux software RAID-5 layer and the SCSI driver to track several statistics. First, we record the amount of time between the first write issued for a stripe and the last write issued for a stripe. This measures the difference between times 1 and 2 in Figure 1, and corresponds directly to the period of vulnerability under the machine failure model.

Second, we record the amount of time between the first write completion for a stripe and the last write completion for a stripe. This measures the difference between time 4 and time 6 in our example. Note, however, that the vulnerability under the disk failure model occurs between time 3 and time 5, so our measurement is an approximation. Our results may slightly overestimate or underestimate the actual vulnerability depending on the

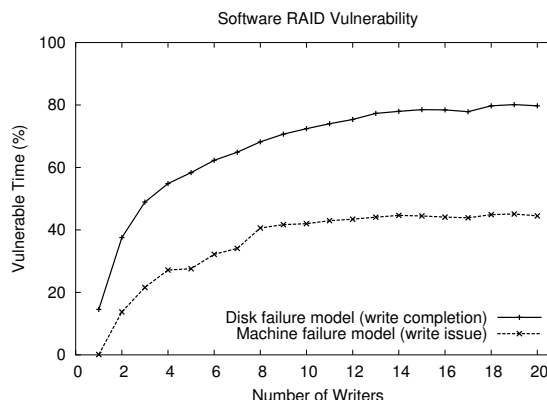


Figure 2: **Software RAID Vulnerability.** The graph plots the percent of time (over the duration of the experiment) that an inconsistent disk state exists in the RAID-5 array as the number of writers increases along the x-axis. Vulnerabilities due to disk failure and machine failure are plotted separately.

time it takes each completion to be sent to and processed by the host machine. Finally, we track the number of stripes that are vulnerable for each of the models. This allows us to calculate the percent of time that any stripe in the array is vulnerable to either type of failure.

Our test workload consists of multiple threads performing synchronous, random writes to a set of files on the array. All of our experiments are performed on an Intel Pentium Xeon 2.6 GHz processor with 512 MB of RAM running Linux kernel 2.6.11. The machine has five IBM 9LZX disks configured as a 1 GB software RAID-5 array. The RAID volume is sufficiently large to perform our benchmarks yet small enough to reduce the execution time of our resynchronization experiments.

Figure 2 plots the percent of time (over the duration of the experiment) that any array stripe is vulnerable as the number of writers in the workload is increased along the x-axis. As expected, the cumulative window of vulnerability increases as the amount of concurrency in the workload is increased. The vulnerability under the disk failure model is greater because it is dependent on the response time of the write requests. Even for a small number of writers, it is more than likely that a disk failure will result in an inconsistent state. For higher concurrency, the array exists in a vulnerable state for up to 80% of the length of the experiment.

The period of vulnerability under the machine failure model is lower because it depends only on the processing time needed to issue the write requests. In our experiment, vulnerability reaches approximately 40%. At much higher concurrencies, however, the ability to issue requests could be impeded by full disk queues. In this case, the machine vulnerability will also depend on the disk response time and will increase accordingly.

2.4 Solutions

To solve this problem, high-end RAID systems make use of non-volatile storage, such as NVRAM. When a write request is received, a log of the request and the data are first written to NVRAM, and then the updates are propagated to the disks. In the event of a crash, the log records and data present in the NVRAM can be used to replay the writes to disk, thus ensuring a consistent state across the array. This functionality comes at an expense, not only in terms of raw hardware, but in the cost of developing and testing a more complex system.

Software RAID, on the other hand, is frequently employed in commodity systems that lack non-volatile storage. When such a system reboots from a crash, there is no record of write activity in the array, and therefore no indication of where RAID inconsistencies may exist. Linux software RAID rectifies this situation by laboriously reading the contents of the entire array, checking the redundant information, and correcting any discrepancies. For RAID-1, this means reading both data mirrors, comparing their contents, and updating one if their states differ. Under a RAID-5 scheme, each stripe of data must be read and its parity calculated, checked against the parity on disk, and re-written if it is incorrect.

This approach fundamentally affects both reliability and availability. The time-consuming process of scanning the entire array lengthens the window of vulnerability during which inconsistent redundancy may lead to data loss under a disk failure. Additionally, the disk bandwidth devoted to resynchronization has a deleterious effect on the foreground traffic serviced by the array. Consequently, there exists a fundamental tension between the demands of reliability and availability: allocating more bandwidth to recover inconsistent disk state reduces the availability of foreground services, but giving preference to foreground requests increases the time to resynchronize.

As observed by Brown and Patterson [2], the default Linux policy addresses this trade-off by favoring availability over reliability, limiting resynchronization bandwidth to 1000 KB/s per disk. Unfortunately, such a slow rate may equate to days of repair time and vulnerability for even moderately sized arrays of hundreds of gigabytes. Figure 3 illustrates this problem by plotting an analytical model of the resynchronization time for a five disk array as the raw size of the array increases along the x-axis. With five disks, the default Linux policy will take almost four minutes of time to scan and repair each gigabyte of disk space, which equates to *two and a half days* for a terabyte of capacity. Disregarding the availability of the array, even modern interconnects would need approximately an hour at their full bandwidth to resynchronize the same one terabyte array.

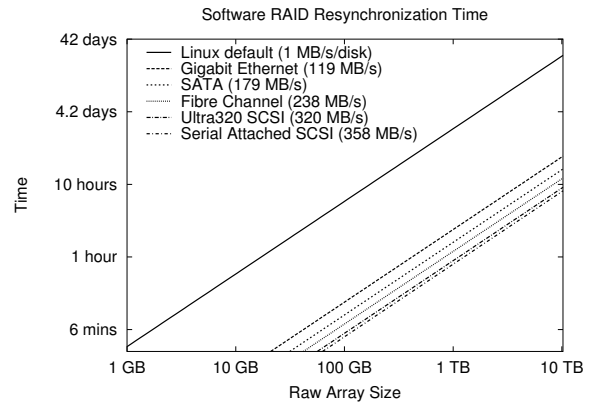


Figure 3: **Software RAID Resynchronization Time.** The graph plots the time to resynchronize a five disk array as the raw capacity increases along the x-axis.

One possible solution to this problem is to add logging to the software RAID system in a manner similar to that discussed above. This approach suffers from two drawbacks, however. First, logging to the array disks themselves would likely decrease the overall performance of the array by interfering with foreground requests. The high-end solution discussed previously benefits from fast, independent storage in the form of NVRAM. Second, adding logging and maintaining an acceptable level of performance could add considerable complexity to the software. For instance, the Linux software RAID implementation uses little buffering, discarding stripes when their operations are complete. A logging solution, however, may need to buffer requests significantly in order to batch updates to the log and improve performance.

Another solution is to perform intent logging to a bitmap representing regions of the array. This mechanism is used by the Solaris Volume Manager [14] and the Veritas Volume Manager [19] to provide optimized resynchronization. An implementation for Linux software RAID-1 is also in development [3], though it has not been merged into the main kernel. Like logging to the array, this approach is likely to suffer from poor performance. For instance, the Linux implementation performs a synchronous write to the bitmap before updating data in the array to ensure proper resynchronization. Performance may be improved by increasing the bitmap granularity, but this comes at the cost of performing scan-based resynchronization over larger regions.

Software RAID is just one layer in the storage hierarchy. One likely configuration contains a modern journaling file system in the layer above, logging disk updates to maintain consistency across its on-disk data structures. In the next sections, we examine how a journaling file system can be used to solve the software RAID resynchronization problem.

3 ext3 Background

In this section, we discuss the Linux ext3 file system, its operation, and its data structures. These details will be useful in our analysis of its write activity and the description of our modifications to support journal-guided resynchronization in Section 4. Although we focus on ext3, we believe our techniques are general enough to apply to other journaling file systems, such as ReiserFS and JFS for Linux, and NTFS for Windows.

Linux ext3 is a modern journaling file system that aims to keep complex on-disk data structures in a consistent state. To do so, all file system updates are first written to a log called the journal. Once the journal records are stored safely on disk, the updates can be applied to their home locations in the main portion of the file system. After the updates are propagated, the journal records are erased and the space they occupied can be re-used.

This mechanism greatly improves the efficiency of crash recovery. After a crash, the journal is scanned and outstanding updates are replayed to bring the file system into a consistent state. This approach constitutes a vast improvement over the previous process (*i.e.* fsck [7]) that relied on a full scan of the file system data structures to ensure consistency. It seems natural, then, to make use of the same journaling mechanism to improve the process of RAID resynchronization after a crash.

3.1 Modes

The ext3 file system offers three modes of operation: data-journaling mode, ordered mode, and writeback mode. In data-journaling mode, all data and metadata is written to the journal, coordinating all updates to the file system. This provides very strong consistency semantics, but at the highest cost. All data written to the file system is written twice: first to the journal, then to its home location.

Ordered mode, the ext3 default, writes all file system metadata to the journal, but file data is written directly to its home location. In addition, this mode guarantees a strict ordering between the writes: all file data for a transaction is written to disk before the corresponding metadata is written to the journal and committed. This guarantees that file metadata will never reference a data block before it has been written. Thus, this mechanism provides the same strong consistency as data-journaling mode without the expense of multiple writes for file data.

In writeback mode, only file system metadata is written to the journal. Like ordered mode, file data is written directly to its home location; unlike ordered mode, however, writeback mode provides no ordering guarantees between metadata and data, therefore offering much weaker consistency. For instance, the metadata for a file

creation may be committed to the journal before the file data is written. In the event of a crash, journal recovery will restore the file metadata, but its contents could be filled with arbitrary data. We will not consider writeback mode for our purposes because of its weaker consistency and its lack of write ordering.

3.2 Transaction Details

To reduce the overhead of file system updates, sets of changes are grouped together into compound transactions. These transactions exist in several phases over their lifetimes. Transactions start in the *running* state. All file system data and metadata updates are associated with the current running transaction, and the buffers involved in the changes are linked to the in-memory transaction data structure. In ordered mode, data associated with the running transaction may be written at any time by the kernel `pdflush` daemon, which is responsible for cleaning dirty buffers. Periodically, the running transaction is closed and a new transaction is started. This may occur due to a timeout, a synchronization request, or because the transaction has reached a maximum size.

Next, the closed transaction enters the *commit* phase. All of its associated buffers are written to disk, either to their home locations or to the journal. After all of the transaction records reside safely in the journal, the transaction moves to the *checkpoint* phase, and its data and metadata are copied from the journal to their permanent home locations. If a crash occurs before or during the checkpoint of a committed transaction, it will be checkpointed again during the journal *recovery* phase of mounting the file system. When the checkpoint phase completes, the transaction is removed from the journal and its space is reclaimed.

3.3 Journal Structure

Tracking the contents of the journal requires several new file system structures. A journal superblock stores the size of the journal file, pointers to the head and tail of the journal, and the sequence number of the next expected transaction. Within the journal, each transaction begins with a *descriptor* block that lists the permanent block addresses for each of the subsequent data or metadata blocks. More than one descriptor block may be needed depending on the number of blocks involved in a transaction. Finally, a *commit* block signifies the end of a particular transaction. Both descriptor blocks and commit blocks begin with a magic header and a sequence number to identify their associated transaction.

4 Design and Implementation

The goal of resynchronization is to correct any RAID inconsistencies that result from system crash or failure. If we can identify the outstanding write requests at the time of the crash, we can significantly narrow the range of blocks that must be inspected. This will result in faster resynchronization and improved reliability and availability. Our hope is to recover such a record of outstanding writes from the file system journal. To this end, we begin by examining the write activity generated by each phase of an ext3 transaction.

4.1 ext3 Write Analysis

In this section, we examine each of the ext3 transaction operations in detail. We emphasize the write requests generated in each phase, and we characterize the possible disk states resulting from a crash. Specifically, we classify each write request as targeting a known location, an unknown location, or a bounded location, based on its record of activity in the journal. Our goal, upon restarting from a system failure, is to recover a record of *all outstanding write requests* at the time of the crash.

Running:

1. In ext3 ordered mode, the `pdflush` daemon may write dirty pages to disk while the transaction is in the running state. If a crash occurs in this state, the affected locations will be unknown, as *no record of the ongoing writes will exist in the journal*.

Commit:

1. ext3 writes all un-journalled dirty data blocks associated with the transaction to their home locations, and waits for the I/O to complete. This step applies only to ordered mode, since all data in data-journaling mode is destined for the journal. If a crash occurs during this phase, the locations of any outstanding writes will be unknown.
2. ext3 writes descriptors, journalled data, and metadata blocks to the journal, and waits for the writes to complete. In ordered mode, only metadata blocks will be written to the journal, whereas all blocks are written to the journal in data-journaling mode. If the system fails during this phase, no specific record of the ongoing writes will exist, but all of the writes will be bounded within the fixed location journal.
3. ext3 writes the transaction commit block to the journal, and waits for its completion. In the event of a crash, the outstanding write is again bounded within the journal.

Block Type	Data-journaling Mode
superblock	known, fixed location
journal	bounded, fixed location
home metadata	known, journal descriptors
home data	known, journal descriptors

Block Type	Ordered Mode
superblock	known, fixed location
journal	bounded, fixed location
home metadata	known, journal descriptors
home data	unknown

Table 1: **Journal Write Records.** *The table lists the block types written during transaction processing and how their locations can be determined after a crash.*

Checkpoint:

1. ext3 writes journalled blocks to their home locations and waits for the I/O to complete. If the system crashes during this phase, the ongoing writes can be determined from the descriptor blocks in the journal, and hence they affect known locations.
2. ext3 updates the journal tail pointer in the superblock to signify completion of the checkpointed transaction. A crash during this operation involves an outstanding write to the journal superblock, which resides in a known, fixed location.

Recovery:

1. ext3 scans the journal checking for the expected transaction sequence numbers (based on the sequence in the journal superblock) and records the last committed transaction.
2. ext3 checkpoints each of the committed transactions in the journal, following the steps specified above. All write activity occurs to known locations.

Table 1 summarizes our ability to locate ongoing writes after a crash for the data-journaling and ordered modes of ext3. In the case of data-journaling mode, the locations of any outstanding writes can be determined (or at least bounded) during crash recovery, be it from the journal descriptor blocks or from the fixed location of the journal file and superblock. Thus, the existing ext3 data-journaling mode is quite amenable to assisting with the problem of RAID resynchronization. On the down side, however, data-journaling typically provides the least performance of the ext3 family.

For ext3 ordered mode, on the other hand, data writes to permanent home locations are not recorded in the journal data structures, and therefore cannot be located dur-

ing crash recovery. We now address this deficiency with a modified ext3 ordered mode: declared mode.

4.2 ext3 Declared Mode

In the previous section we concluded that, if a crash occurs while writing data directly to its permanent location, the ext3 ordered mode journal will contain no record of those outstanding writes. The locations of any RAID level inconsistencies caused by those writes will remain unknown upon restart. To overcome this deficiency, we introduce a new variant of ordered mode, *declared mode*.

Declared mode differs from ordered mode in one key way: it guarantees that a write record for each data block resides safely in the journal before that location is modified. Effectively, the file system must *declare its intent* to write to any permanent location before issuing the write.

To keep track of these intentions, we introduce a new journal block, the *declare* block. A set of declare blocks is written to the journal at the beginning of each transaction commit phase. Collectively, they contain a list of all permanent locations to which data blocks in the transaction will be written. Though their construction is similar to that of descriptor blocks, their purpose is quite different. Descriptor blocks list the permanent locations for blocks that appear in the journal, whereas declare blocks list the locations of blocks that *do not appear* in the journal. Like descriptor and commit blocks, declare blocks begin with a magic header and a transaction sequence number. Declared mode thus adds a single step to the beginning of the commit phase, which proceeds as follows:

Declared Commit:

1. ext3 writes declare blocks to the journal listing each of the permanent data locations to be written as part of the transaction, and it waits for their completion.
2. ext3 writes all un-journalled data blocks associated with the transaction to their home locations, and waits for the I/O to complete.
3. ext3 writes descriptors and metadata blocks to the journal, and waits for the writes to complete.
4. ext3 writes the transaction commit block to the journal, and waits for its completion.

The declare blocks at the beginning of each transaction introduce an additional space cost in the journal. This cost varies with the number of data blocks each transaction contains. In the best case, one declare block will be added for every 506 data blocks, for a space overhead of 0.2%. In the worst case, however, one declare block will be needed for a transaction containing only a single data

block. We investigate the performance consequences of these overheads in Section 5.

Implementing declared mode in Linux requires two main changes. First, we must guarantee that no data buffers are written to disk before they have been declared in the journal. To accomplish this, we refrain from setting the dirty bit on modified pages managed by the file system. This prevents the `pdflush` daemon from eagerly writing the buffers to disk during the running state. The same mechanism is used for all metadata buffers and for data buffers in data-journaling mode, ensuring that they are not written before they are written to the journal.

Second, we need to track data buffers that require declarations, and write their necessary declare blocks at the beginning of each transaction. We start by adding a new *declare tree* to the in-memory transaction structure, and ensure that all declared mode data buffers are placed on this tree instead of the existing *data list*. At the beginning of the commit phase, we construct a set of declare blocks for all of the buffers on the declare tree and write them to the journal. After the writes complete, we simply move all of the buffers from the declare tree to the existing transaction data list. The use of a tree ensures that the writes occur in a more efficient order, sorted by block address. From this point, the commit phase can continue without modification. This implementation minimizes the changes to the shared commit procedure; the other ext3 modes simply bypass the empty declare tree.

4.3 Software RAID Interface

Initiating resynchronization at the file system level requires a mechanism to repair suspected inconsistencies after a crash. A viable option for RAID-1 arrays is for the file system to read and re-write any blocks it has deemed vulnerable. In the case of inconsistent mirrors, either the newly written data or the old data will be restored to each block. This achieves the same results as the current RAID-1 resynchronization process. Because the RAID-1 layer imposes no ordering on mirrored updates, it cannot differentiate new data from old data, and merely chooses one block copy to restore consistency.

This read and re-write strategy is unsuitable for RAID-5, however. When the file system re-writes a single block, our desired behavior is for the RAID layer to calculate its parity across the entire stripe of data. Instead, the RAID layer could perform a read-modify-write by reading the target block and its parity, re-calculating the parity, and writing both blocks to disk. This operation depends on the consistency of the data and parity blocks it reads from disk. If they are not consistent, it will produce incorrect results, simply prolonging the discrepancy. In general, then, a new interface is required for the

file system to communicate possible inconsistencies to the software RAID layer.

We consider two options for the new interface. The first requires the file system to read each vulnerable block and then re-write it with an explicit *reconstruct write* request. In this option, the RAID layer is responsible for reading the remainder of the block's parity group, re-calculating its parity, and then writing the block and the new parity to disk. We are dissuaded from this option because it may perform unnecessary writes to consistent stripes that could cause further vulnerabilities in the event of another crash.

Instead, we opt to add an explicit *verify read* request to the software RAID interface. In this case, the RAID layer reads the requested block along with the rest of its stripe and checks to make sure that the parity is consistent. If it is not, the newly calculated parity is written to disk to correct the problem.

The Linux implementation for the verify read request is rather straight-forward. When the file system wishes to perform a verify read request, it marks the corresponding buffer head with a new *RAID synchronize* flag. Upon receiving the request, the software RAID-5 layer identifies the flag and enables an existing *synchronizing* bit for the corresponding stripe. This bit is used to perform the existing resynchronization process. Its presence causes a read of the entire stripe followed by a parity check, exactly the functionality required by the verify read request.

Finally, an option is added to the software RAID-5 layer to disable resynchronization after a crash. This is our most significant modification to the strict layering of the storage stack. The RAID module is asked to entrust its functionality to another component for the overall good of the system. Instead, an apprehensive software RAID implementation may delay its own efforts in hopes of receiving the necessary verify read requests from the file system above. If no such requests arrive, it could start its own resynchronization to ensure the integrity of its data and parity blocks.

4.4 Recovery and Resynchronization

Using ext3 in either data-journaling mode or declared mode guarantees an accurate view of all outstanding write requests at the time of a crash. Upon restart, we utilize this information and our verify read interface to perform fast, file system guided resynchronization for the RAID layer. Because we make use of the file system journal, and because of ordering constraints between their operations, we combine this process with journal recovery. The dual process of file system recovery and RAID resynchronization proceeds as follows:

Recovery and Resync:

1. ext3 performs verify reads for its superblock and the journal superblock, ensuring their consistency in case they were being written during the crash.
2. ext3 scans the journal checking for the expected transaction sequence numbers (based on the sequence in the journal superblock) and records the last committed transaction.
3. For the first committed transaction in the journal, ext3 performs verify reads for the home locations listed in its descriptor blocks. This ensures the integrity of any blocks undergoing checkpoint writes at the time of the crash. Only the first transaction need be examined because checkpoints must occur in order, and each checkpointed transaction is removed from the journal before the next is processed. Note that these verify reads must take place before the writes are replayed below to guarantee the parity is up-to-date. Adding the explicit reconstruct write interface mentioned earlier would negate the need for this two step process.
4. ext3 issues verify reads beyond the last committed transaction (at the head of the journal) for the length of the maximum transaction size. This corrects any inconsistent blocks as a result of writing the next transaction to the journal.
5. While reading ahead in the journal, ext3 identifies any declare blocks and descriptor blocks for the next uncommitted transaction. If no descriptor blocks are found, it performs verify reads for the permanent addresses listed in each declare block, correcting any data writes that were outstanding at the time of the crash. Declare blocks from transactions containing descriptors can be ignored, as their presence constitutes evidence for the completion of all data writes to permanent locations.
6. ext3 checkpoints each of the committed transactions in the journal as described in Section 4.1.

The implementation re-uses much of the existing framework for the journal recovery process. Issuing the necessary verify reads means simply adding the RAID synchronize flag to the buffers already used for reading the journal or replaying blocks. The verify reads for locations listed in descriptor blocks are handled as the replay writes are processed. The journal verify reads and declare block processing for an uncommitted transaction are performed after the final pass of the journal recovery.

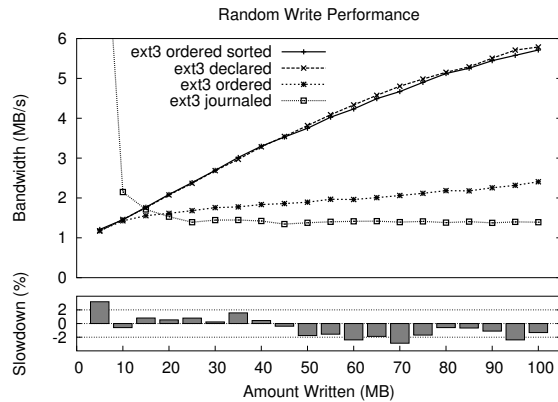


Figure 4: **Random Write Performance.** The top graph plots random write performance as the amount of data written is increased along the x-axis. Data-journaling mode achieves 11.07 MB/s when writing 5 MB of data. The bottom graph shows the relative performance of declared mode as compared to ordered mode with sorting.

5 Evaluation

In this section, we evaluate the performance of ext3 declared mode and compare it to ordered mode and data-journaling mode. We hope that declared mode adds little overhead despite writing extra declare blocks for each transaction. After our performance evaluation, we examine the effects of journal-guided resynchronization. We expect that it will greatly reduce resync time and increase available bandwidth for foreground applications. Finally, we examine the complexity of our implementation.

5.1 ext3 Declared Mode

We begin our performance evaluation of ext3 declared mode with two microbenchmarks, random write and sequential write. First, we test the performance of random writes to an existing 100 MB file. A call to `fsync()` is used at the end of the experiment to ensure that all data reaches disk. Figure 4 plots the bandwidth achieved by each ext3 mode as the amount written is increased along the x-axis. All of our graphs plot the mean of five experimental trials.

We identify two points of interest on the graph. First, data-journaling mode underperforms ordered mode as the amount written increases. Note that data-journaling mode achieves 11.07 MB/s when writing only 5 MB of data because the random write stream is transformed into a large sequential write that fits within the journal. As the amount of data written increases, it outgrows the size of the journal. Consequently, the performance of data-journaling decreases because each block is written twice, first to the journal, and then to its home location. Ordered mode garners better performance by writing data directly to its permanent location.

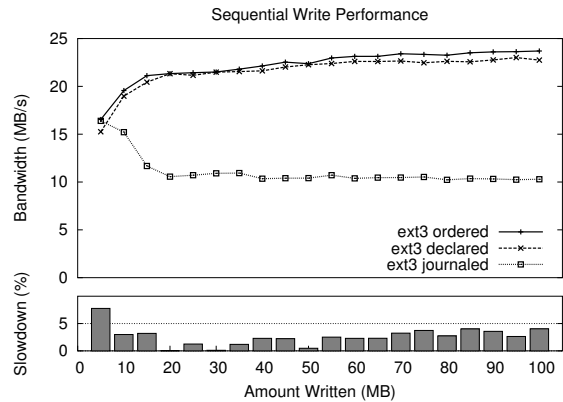


Figure 5: **Sequential Write Performance.** The top graph plots sequential write performance as the amount of data written is increased along the x-axis. The bottom graph shows the relative performance of declared mode as compared to ordered mode.

Second, we find that declared mode greatly outperforms ordered mode as the amount written increases. Tracing the disk activity of ordered mode reveals that part of the data is issued to disk in sorted order based on walking the dirty page tree. The remainder, however, is issued unsorted by the commit phase as it attempts to complete all data writes for the transaction. Adding sorting to the commit phase of ordered mode solves this problem, as evidenced by the performance plotted in the graph. The rest of our performance evaluations are based on this modified version of ext3 ordered mode with sorted writing during commit.

Finally, the bottom graph in Figure 4 shows the slowdown of declared mode relative to ordered mode (with sorting). Overall, the performance of the two modes is extremely close, differing by no more than 3.2%.

Our next experiment tests sequential write performance to an existing 100 MB file. Figure 5 plots the performance of the three ext3 modes. Again, the amount written is increased along the x-axis, and `fsync()` is used to ensure that all data reaches disk. Ordered mode and declared mode greatly outperform data-journaling mode, achieving 22 to 23 MB/s compared to just 10 MB/s.

The bottom graph in Figure 5 shows the slowdown of ext3 declared mode as compared to ext3 ordered mode. Declared mode performs quite well, within 5% of ordered mode for most data points. Disk traces reveal that the performance loss is due to the fact that declared mode waits for `fsync()` to begin writing declare blocks and data. Because of this, ordered mode begins writing data to disk slightly earlier than declared mode. To alleviate this delay, we implement an early declare mode that begins writing declare blocks to the journal as soon as possible, that is, as soon as enough data blocks have been modified to fill a declare block. Unfortunately, this

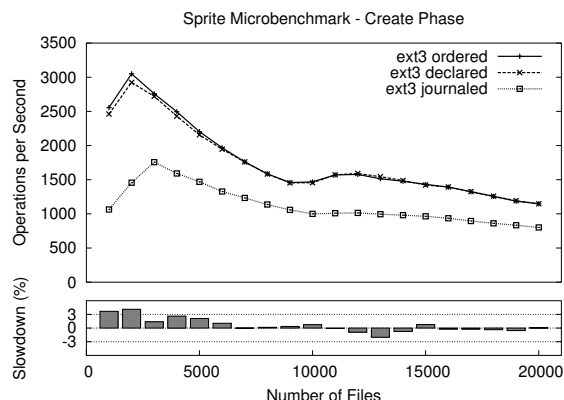


Figure 6: **Sprite Create Performance.** The top graph plots the performance of the create phase of the Sprite LFS microbenchmark as the number of files increases along the x-axis. The bottom graph shows the slowdown of declared mode when compared to ordered mode.

modification does not result in a performance improvement. The early writing of a few declare blocks and data blocks is offset by the seek activity between the journal and the home data locations (not shown).

Next, we examine the performance under the Sprite LFS microbenchmark [10], which creates, reads, and then unlinks a specified number of 4 KB files. Figure 6 plots the number of create operations completed per second as the number of files is increased along the x-axis. The bottom graph shows the slowdown of declared mode relative to ordered mode. Declared mode performs well, within 4% of ordered mode for all cases. The performance of declared mode and ordered mode are nearly identical for the other phases of the benchmark.

The ssh benchmark unpacks, configures, and builds version 2.4.0 of the ssh program from a tarred and compressed distribution file. Figure 7 plots the performance of each mode during the three stages of the benchmark. The execution time of each stage is normalized to that of ext3 ordered mode, and the absolute times in seconds are listed above each bar. Data-journaling mode is slightly faster than ordered mode for the configure phase, but it is 12% slower during build and 378% slower during unpack. Declared mode is quite comparable to ordered mode, running about 3% faster during unpack and configure, and 0.1% slower for the build phase.

Next, we examine ext3 performance on a modified version of the postmark benchmark that creates 5000 files across 71 directories, performs a specified number of transactions, and then deletes all files and directories. Our modification involves the addition of a call to sync() after each phase of the benchmark to ensure that data is written to disk. The unmodified version exhibits unusually high variances for all three modes of operation.

The execution time for the benchmark is shown in Fig-

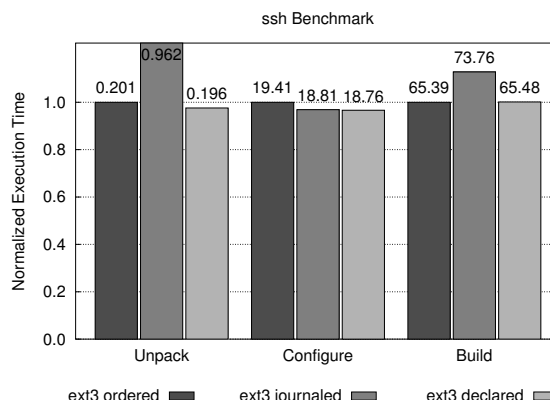


Figure 7: **ssh Benchmark Performance.** The graph plots the normalized execution time of the unpack, configure, and build phases of the ssh benchmark as compared to ext3 ordered mode. The absolute execution times in seconds are listed above each bar.

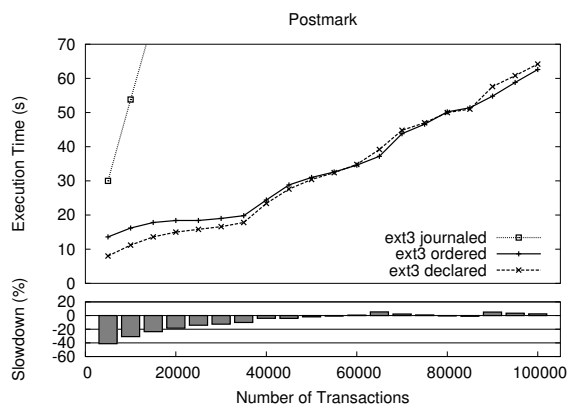


Figure 8: **Postmark Performance.** The top graph plots the execution time of the postmark benchmark as the number of transactions increases along the x-axis. The bottom graph shows the slowdown of declared mode when compared to ordered mode.

ure 8 as the number of transactions increases along the x-axis. Data-journaling mode is extremely slow, and therefore we concentrate on the other two modes, for which we identify two interesting points. First, for large numbers of transactions, declared mode compares favorably to ordered mode, differing by approximately 5% in the worst cases. Second, with a small number of transactions, declared mode outperforms ordered mode by up to 40%. Again, disk traces help to reveal the reason. Ordered mode relies on the sorting provided by the per-file dirty page trees, and therefore its write requests are scattered across the disk. In declared mode, however, the sort performed during commit has a global view of all data being written for the transaction, thus sending the write requests to the device layer in a more efficient order.

Finally, we examine the performance of a TPC-B-like workload that performs a financial transaction across

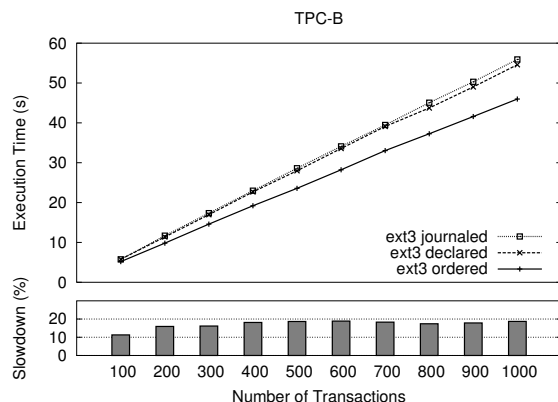


Figure 9: **TPC-B Performance.** The top graph plots the execution time of the TPC-B benchmark as the number of transactions increases along the x-axis. The bottom graph shows the slowdown of declared mode as compared to ordered mode.

three files, adds a history record to a fourth file, and commits the changes to disk by calling `sync()`. The execution time of the benchmark is plotted in Figure 9 as the number of transactions is increased along the x-axis. In this case, declared mode consistently underperforms ext3 ordered mode by approximately 19%, and data-journaling mode performs slightly worse.

The highly synchronous nature of this benchmark presents a worst case scenario for declared mode. Each TPC-B transaction results in a very small ext3 transaction containing only four data blocks, a descriptor block, a journaled metadata block, and a commit block. The declare block at the beginning of each transaction adds 14% overhead in the number of writes performed during the benchmark. To compound this problem, the four data writes are likely serviced in parallel by the array of disks, accentuating the penalty for the declare blocks.

To examine this problem further, we test a modified version of the benchmark that forces data to disk less frequently. This has the effect of increasing the size of each application level transaction, or alternatively simulating concurrent transactions to independent data sets. Figure 10 shows the results of running the TPC-B benchmark with 500 transactions as the interval between calls to `sync()` increases along the x-axis. As the interval increases, the performance of declared mode and data-journaling mode quickly converge to that of ordered mode. Declared mode performs within 5% of ordered mode for `sync()` intervals of five or more transactions.

In conclusion, we find that declared mode routinely outperforms data-journaling mode. Its performance is quite close to that of ordered mode, within 5% (and sometimes better) for our random write, sequential write, and file creation microbenchmarks. It also performs within 5% of ordered mode for two macrobenchmarks,

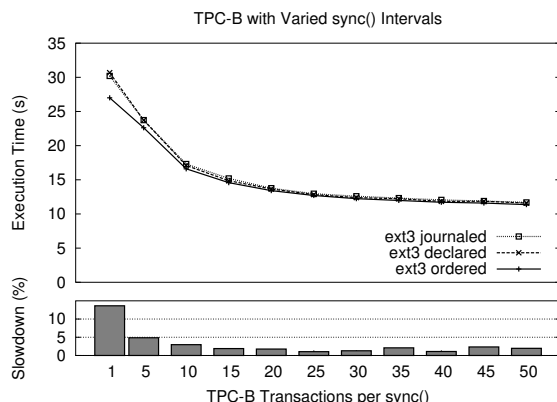


Figure 10: **TPC-B with Varied `sync()` Intervals.** The top graph plots the execution time of the TPC-B benchmark as the interval between calls to `sync()` increases along the x-axis. The bottom graph shows the slowdown of declared mode as compared to ordered mode.

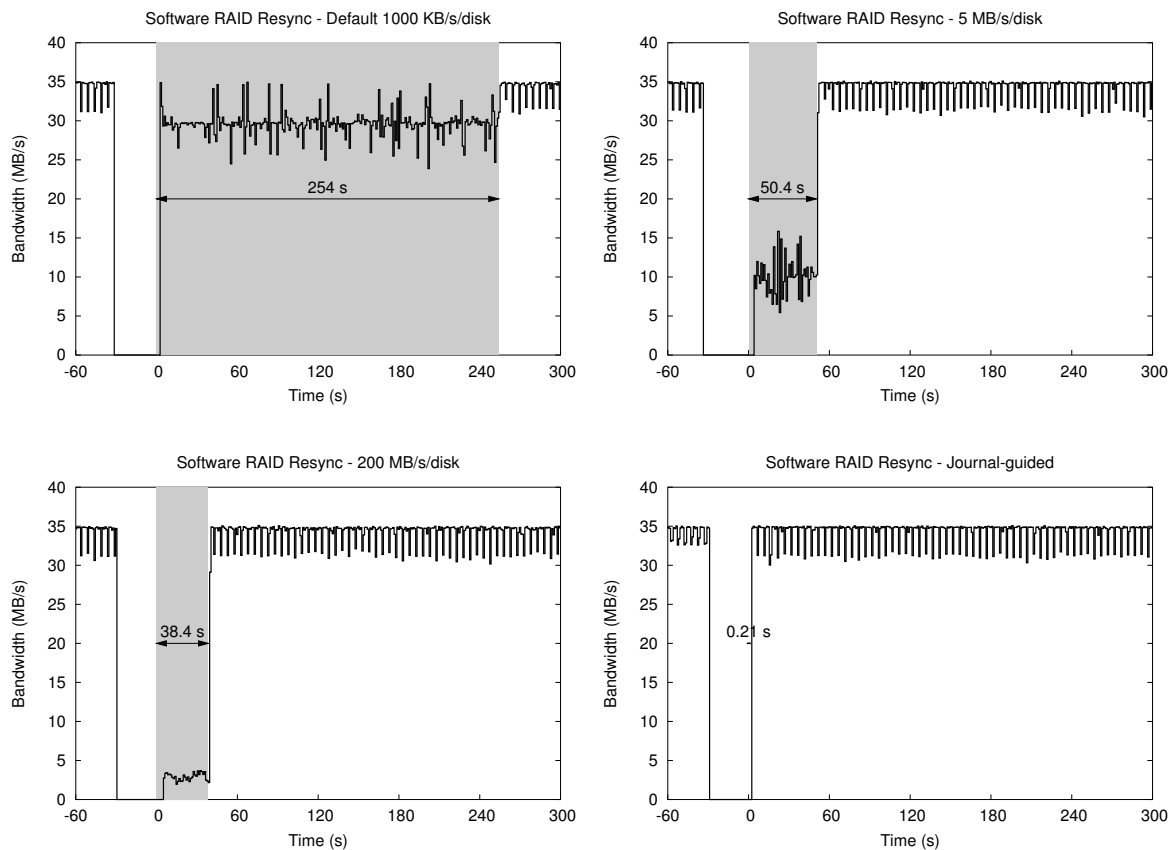
ssh and postmark. The worst performance for declared mode occurs under TPC-B with small application-level transactions, but it improves greatly as the effective transaction size increases. Overall, these results indicate that declared mode is an attractive option for enabling journal-guided resynchronization.

5.2 Journal-guided Resynchronization

In our final set of experiments, we examine the effect of journal-guided resynchronization. We expect a significant reduction in resync time, thus shortening the window of vulnerability and improving reliability. In addition, faster resynchronization should increase the amount of bandwidth available to foreground applications after a crash, thus improving their availability. We compare journal-guided resynchronization to the Linux software RAID resync at the default rate and at two other rates along the availability versus reliability spectrum.

The experimental workload consists of a single foreground process performing sequential reads to a set of large files. The amount of read bandwidth it achieves is measured over one second intervals. Approximately 30 seconds into the experiment, the machine is crashed and rebooted. When the machine restarts, the RAID resynchronization process begins, and the foreground process reactivates as well.

Figure 11 shows a series of such experiments plotting the foreground bandwidth on the y-axis as time progresses on the x-axis. Note that the origin for the x-axis coincides with the beginning of resynchronization, and the duration of the process is shaded in grey. The top left graph in the figure shows the results for the default Linux resync limit of 1000 KB/s per disk, which prefers availability over reliability. The process takes 254 seconds



Resync Type	Resync Rate Limit	Foreground Bandwidth	Vulnerability Window	Vulnerability vs. Default
Default	1000 KB/s/disk	29.58 ± 1.69 MB/s	254.00 s	100.00%
Medium	5 MB/s/disk	29.70 ± 9.48 MB/s	50.41 s	19.84%
High	200 MB/s/disk	29.87 ± 10.65 MB/s	38.44 s	15.13%
Journal-guided		34.09 ± 1.51 MB/s	0.21 s	0.08%

Figure 11: Software RAID Resynchronization. The graphs plot the bandwidth achieved by a foreground process performing sequential scans of files on a software RAID array during a system crash and the ensuing array resynchronization. The recovery period is highlighted in grey and its duration is listed. In the first three graphs, the bandwidth allocated to resynchronization is varied: the default of 1000 KB/s per disk, 5 MB/s per disk, and 200 MB/s per disk. The final graph depicts recovery using journal guidance. The table lists the availability of the foreground service and the vulnerability of the array compared to the default resynchronization period of 254 seconds following restart.

to scan the 1.25 GB of raw disk space in our RAID-5 array. During that time period, the foreground process bandwidth drops to 29 MB/s from the unimpeded rate of 34 MB/s. After resynchronization completes, the foreground process receives the full bandwidth of the array.

Linux allows the resynchronization rate to be adjusted via a sysctl variable. The top right graph in Figure 11 shows the effect of raising the resync limit to 5 MB/s per disk, representing a middle ground between reliability and availability. In this case, resync takes only 50.41 seconds, but the bandwidth afforded the foreground activity drops to only 9.3 MB/s. In the bottom left graph, the resync rate is set to 200 MB/s per disk, favoring reliability over availability. This has the effect of reducing the resync time to 38.44 seconds, but the foreground bandwidth drops to just 2.6 MB/s during that period.

The bottom right graph in the figure demonstrates the use of journal-guided resynchronization. Because of its knowledge of write activity before the crash, it performs much less work to correct any array inconsistencies. The process finishes in just 0.21 seconds, greatly reducing the window of vulnerability present with the previous approach. When the foreground service activates, it has immediate access to the full bandwidth of the array, increasing its availability.

The results of the experiments are summarized in the table in Figure 11. Each metric is calculated over the 254 second period following the restart of the machine in order to compare to the default Linux resynchronization. The 5 MB/s and 200 MB/s resync processes sacrifice availability (as seen in the foreground bandwidth variability) to improve the reliability of the array, reducing the vulnerability windows to 19.84% and 15.13% of the default, respectively. The journal-guided resync process, on the other hand, improves both the availability of the foreground process and the reliability of the array, reducing its vulnerability to just 0.08% of the default case.

It is important to note here that the execution time of the scan-based approach scales linearly with the raw size of the array. Journal-guided resynchronization, on the other hand, is dependent only on the size of the journal, and therefore we expect it to complete in a matter of seconds even for very large arrays.

5.3 Complexity

Table 2 lists the lines of code, counted by the number of semicolons and braces, that were modified or added to the Linux software RAID, ext3 file system, and journaling modules. Very few modifications were needed to add the verify read interface to the software RAID module because the core functionality already existed and merely needed to be activated for the requested stripe. The ext3 changes involved hiding dirty buffers for declared mode

Module	Orig. Lines	Mod. Lines	New Lines	Percent Change
Software RAID	3475	2	16	0.52%
ext3	8621	22	47	0.80%
Journaling	3472	43	265	8.87%
Total	15568	67	328	2.53%

Table 2: **Complexity of Linux Modifications.** *The table lists the lines of code (counting semicolons and braces) in the original Linux 2.6.11 source and the number that were modified or added to each of the software RAID, ext3 file system, and journaling modules.*

and using verify reads during recovery. The majority of the changes occurred in the journaling module for writing declare blocks in the commit phase and performing careful resynchronization during recovery.

As a point of comparison, the experimental version of Linux RAID-1 bitmap logging consists of approximately 1200 lines of code, a 38% increase over RAID-1 alone. Most of our changes are to the journaling module, increasing its size by about 9%. Overall, our modifications consist of 395 lines of code, a 2.5% change across the three modules. These observations support our claim that leveraging functionality across cooperating layers can reduce the complexity of the software system.

6 Related Work

Brown and Patterson [2] examine three different software RAID systems in their work on availability benchmarks. They find that the Linux, Solaris, and Windows implementations offer differing policies during reconstruction, the process of regenerating data and parity after a disk failure. Solaris and Windows both favor reliability, while the Linux policy favors availability. Unlike our work, the authors do not focus on improving the reconstruction processes, but instead on identifying their characteristics via a general benchmarking framework.

Stodolsky *et al.* [13] examine parity logging in the RAID layer to improve the performance of small writes. Instead of writing new parity blocks directly to disk, they store a log of parity update images which are batched and written to disk in one large sequential access. Similar to our discussion of NVRAM logging, the authors require the use of a fault tolerant buffer to store their parity update log, both for reliability and performance. These efforts to avoid small random writes support our argument that maintaining performance with RAID level logging is a complex undertaking.

The Veritas Volume Manager [19] provides two facilities to address faster resynchronization. A dirty region log can be used to speed RAID-1 resynchronization by

examining only those regions that were active before a crash. Because the log requires extra writes, however, the author warns that coarse-grained regions may be needed to maintain acceptable write performance. The Volume Manager also supports RAID-5 logging, but non-volatile memory or a solid state disk is recommended to support the extra log writes. In contrast, our declared mode offers fine-grained journal-guided resynchronization with little performance degradation and without the need for additional hardware.

Schindler *et al.* [11] augment the RAID interface to provide information about individual disks. Their Atropos volume manager exposes disk boundary and track information to provide efficient semi-sequential access to two-dimensional data structures such as database tables. Similarly, E×RAID [4] provides disk boundary and performance information to augment the functionality of an informed file system. Our verify read interface is much less complex, providing file system access to functionality that already exists in the software RAID layer.

7 Conclusions

We have examined the ability of a journaling file system to provide support for faster software RAID resynchronization. In order to obtain a record of the outstanding writes at the time of a crash, we introduce ext3 declared mode. This new mode guarantees to declare its intentions in the journal before writing data to disk. Despite this extra write activity, declared mode performs within 5% of its predecessor.

In order to communicate this information to the software RAID layer, the file system utilizes a new verify read request. This request instructs the RAID layer to read the block and repair its redundant information, if necessary. Combining these features allows us to implement fast, journal-guided resynchronization. This process improves both software RAID reliability and availability by hastening the recovery process after a crash.

Our general approach advocates a system-level view for developing the storage stack. Using the file system journal to improve the RAID system leverages existing functionality, maintains performance, and avoids duplicating complexity at multiple layers. Each of these layers may implement its own abstractions, protocols, mechanisms, and policies, but it is often their interactions that define the properties of a system.

8 Acknowledgements

We would like to thank John Bent, Nathan Burnett, and the anonymous reviewers for their excellent feedback. This work is sponsored by NSF CCR-0092840, CCR-0133456, NGS-0103670, ITR-0325267, Network Appliance, and EMC.

References

- [1] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.
- [2] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.
- [3] P. Clements and J. Bottomley. High Availability Data Replication. In *Proceedings of the 2003 Linux Symposium*, Ottawa, ON, Canada, June 2003.
- [4] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, California, June 2002.
- [5] EMC. EMC Centera: Content Addressed Storage System. <http://www.emc.com/>, 2004.
- [6] EMC Corporation. Symmetrix Enterprise Information Storage Systems. <http://www.emc.com>, 2002.
- [7] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsync - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.
- [8] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.
- [9] H. Reiser. ReiserFS. www.namesys.com, 2004.
- [10] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.
- [11] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.
- [12] D. A. Solomon. *Inside Windows NT (Microsoft Programming Series)*. Microsoft Press, 1998.
- [13] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, pages 64–75, San Diego, California, May 1993.
- [14] Sun. Solaris Volume Manager Administration Guide. <http://docs.sun.com/app/docs/doc/816-4520>, July 2005.
- [15] D. Teigland and H. Mauelshagen. Volume Managers in Linux. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Boston, Massachusetts, June 2001.
- [16] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.
- [17] S. C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.
- [18] S. C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.
- [19] Veritas. Features of VERITAS Volume Manager for Unix and VERITAS File System. <http://www.veritas.com/us/products/volumemanager/whitepaper-02.html>, July 2005.

DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality

Song Jiang
*Performance & Architecture Laboratory
Computer & Computational Sciences Div.
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
sjiang@lanl.gov*

Xiaoning Ding, Feng Chen,
Enhua Tan and Xiaodong Zhang
*Department of Computer Science and Engineering
Ohio State University
Columbus, OH 43210, USA
{dingxn, fchen, etan, zhang}@cse.ohio-state.edu*

Abstract

Sequentiality of requested blocks on disks, or their spatial locality, is critical to the performance of disks, where the throughput of accesses to sequentially placed disk blocks can be an order of magnitude higher than that of accesses to randomly placed blocks. Unfortunately, spatial locality of cached blocks is largely ignored and only temporal locality is considered in system buffer cache management. Thus, disk performance for workloads without dominant sequential accesses can be seriously degraded. To address this problem, we propose a scheme called *DULO* (*DUal LOcality*), which exploits both temporal and spatial locality in buffer cache management. Leveraging the filtering effect of the buffer cache, *DULO* can influence the I/O request stream by making the requests passed to disk more sequential, significantly increasing the effectiveness of I/O scheduling and prefetching for disk performance improvements.

DULO has been extensively evaluated by both trace-driven simulations and a prototype implementation in Linux 2.6.11. In the simulations and system measurements, various application workloads have been tested, including Web Server, TPC benchmarks, and scientific programs. Our experiments show that *DULO* can significantly increase system throughput and reduce program execution times.

1 Introduction

A hard disk drive is the most commonly used secondary storage device supporting file accesses and virtual memory paging. While its capacity growth pleasantly matches the rapidly increasing data storage demand, its electromechanical nature causes its performance improvements to lag painfully far behind processor speed progress. It is apparent that the disk bottleneck effect is worsening in modern computer systems, while the role of the hard disk as dominant storage device will not change in the foreseeable future, and the amount of

disk data requested by applications continues to increase.

The performance of a disk is limited by its mechanical operations, including disk platter rotation (*spinning*) and disk arm movement (*seeking*). A disk head has to be on the right track through seeking and on the right sector through spinning for reading/writing its desired data. Between the two moving components of a disk drive affecting its performance, the disk arm is its Achilles' Heel. This is because an actuator has to move the arm accurately to the desired track through a series of actions including acceleration, coast, deceleration, and settle. Thus, accessing a stream of sequential blocks on the same track achieves a much higher disk throughput than that accessing several random blocks does.

In current practice, there are several major efforts in parallel to break the disk bottleneck. One effort is to reduce disk accesses through memory caching. By using replacement algorithms to exploit the temporal locality of data accesses, where data are likely to be re-accessed in the near future after they are accessed, disk access requests can be satisfied without actually being passed to disk. To minimize disk activities in the number of requested blocks, all the current replacement algorithms are designed by adopting block miss reduction as the sole objective. However, this can be a misleading metric that may not accurately reflect real system performance. For example, requesting ten sequential disk blocks can be completed much faster than requesting three random disk blocks, where disk seeking is involved. To improve real system performance, spatial locality, a factor that can make a difference as large as an order of magnitude in disk performance, must be considered. However, spatial locality is unfortunately ignored in current buffer cache management. In the context of this paper, spatial locality specifically refers to the sequentiality of continuously requested blocks' disk placements.

Another effort to break the disk bottleneck is reducing disk arm seeks through I/O request scheduling. I/O scheduler reorders pending requests in a block device's re-

quest queue into a dispatching order that results in minimal seeks and thereafter maximal global disk throughput. Example schedulers include Shortest-Seek-Time-First (SSTF), CSCAN, as well as the Deadline and Anticipatory I/O schedulers [15] adopted in the current Linux kernel.

The third effort is prefetching. The data prefetching manager predicts the future request patterns associated with a file opened by a process. If a sequential access pattern is detected, then the prefetching manager issues requests for the blocks following the current on-demand block on behalf of the process. Because a file is usually continuously allocated on disk, these prefetching requests can be fulfilled quickly with few disk seeks.

While I/O scheduling and prefetching can effectively exploit spatial locality and dramatically improve disk throughput for workloads with dominant sequential accesses, their ability to deal with workloads mixed with sequential and random data accesses, such as those in Web services, databases, and scientific computing applications, is very limited. This is because these two schemes are positioned at a level lower than the buffer cache. While the buffer cache receives I/O requests directly from applications and has the power to shape the requests into a desirable I/O request stream, I/O scheduling and prefetching only work on the request stream passed on by the buffer cache and have very limited ability to recatch the opportunities lost in buffer cache management. Hence, in the worst case, a stream filled with random accesses prevents I/O scheduling and prefetching from helping, because no spatial locality is left for them to exploit.

Concerned with the missing ability to exploit spatial locality in buffer cache management, our solution to the deteriorating disk bottleneck is a new buffer cache management scheme that exploits both temporal and spatial locality, which we call the *DUal LOcality* scheme *DULO*. *DULO* introduces dual locality into the caching component in the OS by tracking and utilizing the disk placements of in-memory pages in buffer cache management¹. Our objective is to maximize the sequentiality of I/O requests that are served by disks. For this purpose, we give preference to random blocks for staying in the cache, while sequential blocks that have their temporal locality comparable to those random blocks are replaced first. With the filtering effect of the cache on I/O requests, we influence the I/O requests from applications so that more sequential block requests and less random block requests are passed to the disk thereafter. The disk is then able to process the requests with stronger spatial locality more efficiently.

2 Dual Locality Caching

2.1 An Illustrating Example

To illustrate the differences that a traditional caching scheme could make when equipped with dual locality ability, let us

consider an example reference stream mixed with sequential and random blocks. In the accessed blocks, we assume blocks A, B, C, and D are random blocks dispersed across different tracks. Blocks X1, X2, X3, and X4 as well as blocks Y1, Y2, Y3, and Y4 are sequential blocks located on their respective tracks. Furthermore, two different files consist of blocks X1, X2, X3, and X4, and blocks Y1, Y2, Y3 and Y4, respectively. Assume that the buffer cache has room for eight blocks. We also assume that the LRU replacement algorithm and a Linux-like prefetching policy are applied. In this simple illustration, we use the average seek time to represent the cost of any seek operation, and use average rotation time to represent the cost of any rotation operation². We ignore other negligible costs such as disk read time and bus transfer time. The 6.5 ms average seek time and 3.0 ms average rotation time are taken from the specification of the Hitachi Ultrastar 18ZX 10K RPM drive.

Table 1 shows the reference stream and the on-going changes of cache states, as well as the time spent on each access for the traditional caching and prefetching scheme (denoted as *traditional*) and its dual locality conscious alternative (denoted as *dual*). In the 5th access, prefetching is activated and all the four sequential blocks are fetched because the prefetcher knows the reference (to block X1) starts at the beginning of the file. The difference in the cache states between the two schemes here is that *traditional* lists the blocks in the strict LRU order, while *dual* re-arranges the blocks and places the random blocks at the MRU end of the queue. Therefore, the four random blocks A, B, C, and D are replaced in *traditional*, while sequential blocks X1, X2, X3, and X4 are replaced in *dual* when the 9th access incurs a four-block prefetching. The consequences of these two choices are two different miss streams that turn into real disk requests. For *traditional*, it is {A, B, C, D} from the 17th access, a four random block disk request stream, and the total cost is 95.0 ms. For *dual*, it is {X1, X2, X3, X4} at the 13th access, a four sequential blocks, and the total cost is only 66.5 ms.

If we do not enable prefetching, the two schemes have the same number of misses, i.e., 16. With prefetching enabled, *traditional* has 10 misses, while *dual* has only 7 misses. This is because *dual* generates higher quality I/O requests (containing more sequential accesses) to provide more prefetching opportunities.

2.2 Challenges with Dual Locality

Introducing dual locality in cache management raises challenges that do not exist in the traditional system, which is evident even in the above simple illustrating example.

In current cache management, replacement algorithms only consider temporal locality (a position in the queue in the case of LRU) to make a replacement decision. While introducing spatial locality necessarily has to compromise

	Block Being Accessed	<i>Traditional</i>	Time (ms)	<i>Dual</i>	Time (ms)
1	A	[A - - - - -]	9.5	[A - - - - -]	9.5
2	B	[B A - - - - -]	9.5	[B A - - - - -]	9.5
3	C	[C B A - - - -]	9.5	[C B A - - - -]	9.5
4	D	[D C B A - - - -]	9.5	[D C B A - - - -]	9.5
5	X1	[X4 X3 X2 X1 D C B A]	9.5	[D C B A X4 X3 X2 X1]	9.5
6	X2	[X2 X4 X3 X1 D C B A]	0	[D C B A X2 X4 X3 X1]	0
7	X3	[X3 X2 X4 X1 D C B A]	0	[D C B A X3 X2 X4 X1]	0
8	X4	[X4 X3 X2 X1 D C B A]	0	[D C B A X4 X3 X2 X1]	0
9	Y1	[Y4 Y3 Y2 Y1 X4 X3 X2 X1]	9.5	[D C B A Y4 Y3 Y2 Y1]	9.5
10	Y2	[Y2 Y4 Y3 Y1 X4 X3 X2 X1]	0	[D C B A Y2 Y4 Y3 Y1]	0
11	Y3	[Y3 Y2 Y4 Y1 X4 X3 X2 X1]	0	[D C B A Y3 Y2 Y4 Y1]	0
12	Y4	[Y4 Y3 Y2 Y1 X4 X3 X2 X1]	0	[D C B A Y4 Y3 Y2 Y1]	0
13	X1	[X1 Y4 Y3 Y2 Y1 X4 X3 X2]	0	[D C B A X4 X3 X2 X1]	9.5
14	X2	[X2 X1 Y4 Y3 Y2 Y1 X4 X3]	0	[D C B A X2 X4 X3 X1]	0
15	X3	[X3 X2 X1 Y4 Y3 Y2 Y1 X4]	0	[D C B A X3 X2 X4 X1]	0
16	X4	[X4 X3 X2 X1 Y4 Y3 Y2 Y1]	0	[D C B A X4 X3 X2 X1]	0
17	A	[A X4 X3 X2 X1 Y4 Y3 Y2]	9.5	[A D C B X4 X3 X2 X1]	0
18	B	[B A X4 X3 X2 X1 Y4 Y3]	9.5	[B A D C X4 X3 X2 X1]	0
19	C	[C B A X4 X3 X2 X1 Y4]	9.5	[C B A D X4 X3 X2 X1]	0
20	D	[D C B A X4 X3 X2 X1]	9.5	[D C B A X4 X3 X2 X1]	0
		total time	95.0	total time	66.5

Table 1: An example showing that a dual locality conscious scheme can be more effective than its traditional counterpart in improving disk performance. Fetched blocks are boldfaced. The MRU end of the queue is on the left.

the weight of temporal locality in the replacement decision, the role of temporal locality must be appropriately retained in the decision. In the example shown in Table 1, we give random blocks A, B, C, and D more privilege of staying in cache by placing them at the MRU end of the queue due to their weak spatial locality (weak sequentiality), even though they have weak temporal locality (large recency). However, we certainly cannot keep them in cache forever if they have few re-accesses showing sufficient temporal locality. Otherwise, they would pollute the cache with inactive data and reduce the effective cache size. The same consideration also applies to the block sequences of different sizes. We prefer to keep a short sequence because it only has a small number of blocks to amortize the almost fixed cost of an I/O operation. However, how do we make a replacement decision when we encounter a not recently accessed short sequence and a recently accessed long sequence? The challenge is how to make the tradeoff between temporal locality (recency) and spatial locality (sequence size) with the goal of maximizing disk performance.

3 The DULO Scheme

We now present our DULO scheme to exploit both temporal locality and spatial locality simultaneously and seamlessly. Because LRU or its variants are the most widely used replacement algorithms, we build the DULO scheme by using the LRU algorithm and its data structure — the LRU stack, as a reference point.

In LRU, newly fetched blocks enter into its stack top and replaced blocks leave from its stack bottom. There are

two key operations in the DULO scheme: (1) Forming sequences. A *sequence* is defined as a number of blocks whose disk locations are adjacent³ and have been accessed during a limited time period. Because a sequence is formed from the blocks in a stack segment of limited size, and all blocks enter into the stack due to their references, the second condition of the definition is automatically satisfied. Specifically, a random block is a sequence of size 1. (2) Sorting the sequences in the LRU stack according to their recency (temporal locality) and size (spatial locality), with the objective that sequences of large recency and size are placed close to the LRU stack bottom. Because the recency of a sequence is changing while new sequences are being added, the order of the sorted sequence should be adjusted dynamically to reflect the change.

3.1 Structuring LRU stack

To facilitate the operations presented above, we partition the LRU stack into two sections (shown in Figure 1 as a vertically placed queue). The top part is called *staging section* used for admitting newly fetched blocks, and the bottom part is called *evicting section* used for storing sorted sequences to be evicted in their orders. We again divide the staging section into two segments. The first segment is called *correlation buffer*, and the second segment is called *sequencing bank*. The correlation buffer in DULO is similar to the *correlation reference period* used in the LRU-K replacement algorithm [26]. Its role is to filter high frequency references and to keep them from entering the sequencing bank, so as to reduce the consequential operational cost. The size of the

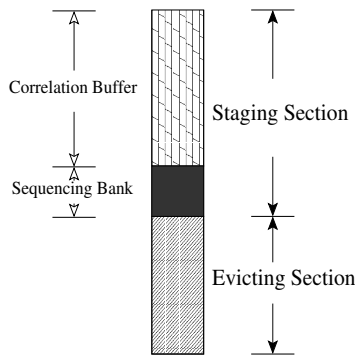


Figure 1: LRU stack is structured for the DULO replacement algorithm.

buffer is fixed. The sequencing bank is used to prepare a collection of blocks to be sequenced, and its size ranges from 0 to a maximum value, *BANK_MAX*.

Suppose we start with an LRU stack whose staging section consists of only the correlation buffer (the size of the sequencing bank is 0), and the evicting section holds the rest of the stack. When a block leaves the evicting section at its bottom and a block enters the correlation buffer at its top, the bottom block of the correlation buffer enters the sequencing bank. When there are *BANK_MAX* blocks leaving the evicting section, the size of sequencing bank is *BANK_MAX*. We refill the evicting section by taking the blocks in the bank, forming sequences out of them, and inserting them into the evicting section in a desired order. There are three reasons for us to maintain two interacting sections and use the bank to conduct sequence forming: (1) The newly admitted blocks have a buffering area to be accumulated for forming potential sequences. (2) The sequences formed at the same time must share a common recency, because their constituent blocks are from the same block pool — the sequencing bank in the staging section. By restricting the bank size, we make sure that the block recency will not be excessively compromised for the sake of spatial locality. (3) The blocks that are leaving the stack are sorted in the evicting section for a replacement order reflecting both their sequentiality and their recency.

3.2 Block Table: A Data Structure for Dual Locality

To complement the missing spatial locality in traditional caching systems, we introduce a data structure in the OS kernel called *block table*. The block table is analogous in structure to the multi-level page table used for process address translation. However there are clear differences between them because they serve different purposes: (1) The page table covers virtual address space of a process in the unit of page and page address is an index into the table, while the block table covers disk space in the unit of block, and block disk location is an index into the table. (2) The page table is used to translate a virtual address into its physical address, while the block table is used to provide the times of

recent accesses for a given disk block. (3) The requirement on the page table lookup efficiency is much more demanding and performance-critical than that on the block table lookup efficiency because the former supports instruction execution while the latter facilitates I/O operations. That is the reason why a hardware TLB has to be used to expedite page table lookup, but there is no such a need for block table. (4) Each process owns a page table, while each disk drive owns a block table in memory.

In the system we set a global variable called *bank clock*, which ticks each time the bank in the staging section is used for forming sequences. Each block in the bank takes the current clock time as a timestamp representing its most recent access time. We then record the timestamp in an entry at the leaf level of the block table corresponding to the block disk location, which we called BTE (*Block Table Entry*). BTE is analogous in structure to PTE (*Page Table Entry*) of page table. Each BTE allows at most two most recent access times recorded in it. Whenever a new time is added, the oldest time is replaced if the BTE is full. In addition, to manage efficiently the memory space held by block table(s), a timestamp is set in each table entry at directory levels (equivalent to PGD (*Page Global Directory*) and PMD (*Page Middle Directory*) in the Linux page table). Each time the block table is looked up in a hierarchical way to record a new access time, the time is also recorded as a timestamp in each directory entry that has been passed. In this way, each directory entry keeps the most recent timestamp among those of all its direct/indirect children entires when the table is viewed as a tree. The entries of the table are allocated in an on-demand fashion.

The memory consumption of the block table can be flexibly controlled. When system memory pressure is too high and the system needs to reclaim memory held by the table, it traverses the table with a specified clock time threshold for reclamation. Because the most recent access times are recorded in the directories, the system will remove a directory once it finds its timestamp is smaller than the threshold, and all the subdirectories and BTEs under it will be removed.

3.3 Forming Sequences

When it is the time to form sequences from a full bank, the bank clock is incremented by one. Each block in the bank then records the clock time as a new timestamp in the block table. After that, we traverse the table to collect all the sequences consisting of the blocks with the current clock time. This is a low cost operation because each directory at any level in a block table contains the most recent timestamp among all the BTEs under it. The traversal goes into only those directories containing the blocks in the bank. To ensure the stability of a sequence exhibited in history, the algorithm determines that the last block of a developing sequence should not be coalesced with the next block in the table if the

next block belongs to one of the following cases:

1. Its BTE shows that it was accessed in the current clock time. This includes the case where it has never been accessed (i.e., it has an empty timestamp field). It belongs to this case if the next block is a spare or defective block on the disk.
2. One and only one of the two blocks, the current block and the next block, was not accessed before the current clock time (i.e., it has only one timestamp).
3. Both of the two blocks have been accessed before the current clock time, but their last access times have a difference exceeding one.
4. The current sequence size reaches 128, which is the maximal allowed sequence size and we deem to be sufficient to amortize a disk operation cost.

If any one of the conditions is met, a complete sequence has been formed and a new sequence starts to be formed. Otherwise, the next block becomes part of the sequence, the following blocks will be tested continuously.

3.4 The DULO Replacement Algorithm

There are two challenging issues to be addressed in the design of the DULO replacement algorithm.

The first issue is the potentially prohibitive overhead associated with the DULO scheme. In the strict LRU algorithm, both missed blocks and hit blocks are required to move to the stack top. This means that a hit on a block in the evicting section is associated with a bank sequencing cost and a cost for sequence ordering in the evicting section. These additional costs that can incur in a system with few memory misses are unacceptable. In fact, the strict LRU algorithm is seldom used in real systems because of its overhead associated with every memory reference [18]. Instead, its variant, the CLOCK replacement algorithm, has been widely used in practice. In CLOCK, when a block is hit, it is only flagged as *young* block without being moved to the stack top. When a block has to be replaced, the block at the stack bottom is examined. If it is a young block, it is moved to the stack top and its “young block” status is revoked. Otherwise, the block is replaced. It is known that CLOCK simulates LRU behaviors very closely and its hit ratios are very close to those of LRU. For this reason, we build the DULO replacement algorithm based on the CLOCK algorithm. That is, it delays the movement of a hit block until it reaches the stack bottom. In this way, only block misses could trigger sequencing and the evicting section refilling operations. While being compared with the miss penalty, these costs are very small.

The second issue is how sequences in the evicting section are ordered for replacement according to their temporal and spatial locality. We adopt an algorithm similar to

```
Initialize L = 0;
losses_of_evicting_section = 0;

/* Procedure to be invoked upon a reference
   to block b */

if b is in cache
    mark b as a young block;
else {
    while (block e at the stack bottom is young) {
        revoke the young block state;
        move it to the stack top;
        losses_of_evicting_section++;
        if (losses_of_evicting_section == BANK_MAX)
            refill_evicting_section();
    }
    replace block e at the stack bottom;
    s = e.sequence;
    L = H(s);
    losses_of_evicting_section++;
    if (losses_of_evicting_section == BANK_MAX)
        refill_evicting_section();
    place block b at the stack top as a young block
}

/* procedure to refill the evicting section */
refill_evicting_section()
{
    /* group sequences */
    for each block in sequencing bank
        place it in hierarchical block table;

    traverse the table to obtain all sequences;
    for each above sequence s
        H(s) = L + 1/size(s);
    sort the above sequences by H(s) into list L1;

    /* L2 is the list of sequences in evicting
       section */
    evicting_section = merge_sort(L1, L2);
    losses_of_evicting_section = 0;
}
```

Figure 2: The DULO Replacement Algorithm

GreedyDual-Size used in Web file caching [8]. *GreedyDual-Size* was originally derived from *GreedyDual* [37]. It makes its replacement decision by considering the recency, size, and fetching cost of cached files. It has been proven that *GreedyDual-Size* is online-optimal, which is k -competitive, where k is the ratio of the size of the cache to the size of the smallest file. In our case, file size is equivalent to sequence size, and file fetching cost is equivalent to the I/O operation cost for a sequence access. For sequences whose sizes are distributed in a reasonable range, which is limited by bank size, we currently assume their fetching cost is the same. Our algorithm can be modified to accommodate cost variance if necessary in the future.

The DULO algorithm associates each sequence with a value H , where a relatively small value indicates the sequence should be evicted first. The algorithm has a global inflation value L , which records the H value of the most

recent evicted sequence. When a new sequence s is admitted into the evicting section, its H value is set as $H(s) = L + 1/\text{size}(s)$, where $\text{size}(s)$ is the number of the blocks contained in s . The sequences in the evicting section are sorted by their H values with sequences of small H values at the LRU stack bottom. In the algorithm a sequence of large size tends to stay at the stack bottom and to be evicted first. However, if a sequence of small size is not accessed for a relatively long time, it will be replaced. This is because a newly admitted long sequence could have a larger H value due to the L value, which is continuously being inflated by the evicted blocks. When all sequences are random blocks (i.e., their sizes are 1), the algorithm degenerates into the LRU replacement algorithm.

As we have mentioned before, once a bank size of blocks are replaced from the evicting section, we take the blocks in the sequencing bank to form sequences and order the sequences by their H values. Note that all these sequences share the same current L value in their H value calculations. With a merge sorting of the newly ordered sequence list and the ordered sequence list in the evicting section, we complete the refilling of the evicting section, and the staging section ends up with only the correlation buffer. The algorithm is described using pseudo code in Figure 2.

4 Performance Evaluation

We use both trace-driven simulations and a prototype implementation to evaluate the DULO scheme and to demonstrate the impact of introducing spatial locality into replacement decisions on different access patterns in applications.

4.1 The DULO Simulation

4.1.1 Experiment Settings

We built a simulator that implements the DULO scheme, Linux prefetching policy [28], and Linux Deadline I/O scheduler [30]. We also interfaced the Disksim 3.0, an accurate disk simulator [4], to simulate the disk behaviors. The disk drive we modeled is the Seagate ST39102LW with 10K RPM and 9.1GB capacity. Its maximum read/write seek time is 12.2/13.2ms, and its average rotation time is 2.99ms. We selected five traces of representative I/O request patterns to drive the simulator (see Table 2). The traces have also been used in [5], where readers are referred for their details. Here we briefly describe these traces.

Trace *viewperf* consists of almost all-sequential-accesses. The trace was collected by running SPEC 2000 benchmark *viewperf*. In this trace, over 99% of its references are to consecutive blocks within a few large files. By contrast, trace *tpc-h* consists of almost all-random-accesses. The trace was collected when the TPC-H decision support benchmark runs on the MySQL database system. TPC-H performs random

references to several large database files, resulting in only 3% references to consecutive blocks in the trace.

The other three traces have mixed I/O request patterns. Trace *glimpse* was collected by using the indexing and query tool “glimpse” to search for text strings in the text files under the */usr* directory. Trace *multi1* was collected by running programs *cscope*, *gcc*, and *viewperf* concurrently. *Cscope* is a source code examination tool, and *gcc* is a GNU compiler. Both take Linux kernel 2.4.20 source code as their inputs. *Cscope* and *glimpse* have a similar access pattern. They contain 76% and 74% sequential accesses, respectively. Trace *multi2* was collected by running programs *glimpse* and *tpc-h* concurrently. *Multi2* has a lower sequential access rate than *Multi1* (16% vs. 75%).

In the simulations, we set the sequencing bank size as 8MB, and evicting section size as 64MB in most cases. Only in the cases where the demanded memory size is less than 80MB (such as for *viewperf*), we set the sequencing bank size as 4MB, and evicting section size as 16MB. These choices are based on the results of our parameter sensitivity studies to be presented in Section 4.1.3. In the evaluation, we compare the DULO performance with that of the CLOCK algorithm. For generality, we still refer it as LRU.

4.1.2 Evaluation Results

Figures 3 and 4 show the execution times, hit ratios, and disk access sequence size distributions of the LRU caching and DULO caching schemes for the five workloads when we vary memory size. Because the major effort of DULO to improve system performance is to influence the quality of the requests presented to the disk — the number of sequential block accesses (or sequence size), we show the sequence size differences for workloads running on the LRU caching scheme and on the DULO caching scheme. For this purpose, we use CDF curves to show how many percentages (shown on Y-axis) of requested blocks appear in the sequences whose sizes are less than a certain threshold (shown on X-axis). For each trace, we select two memory sizes to draw the corresponding CDF curves for LRU and DULO, respectively. We select the memory sizes according to the execution time gaps between LRU and DULO shown in execution time figures — one memory size is selected due to its small gap and another is selected due to its large gap. The memory sizes are shown in the legends of the CDF figures.

First, examine Figure 3. The CDF curves show that for the almost-all-sequential workload *viewperf*, more than 80% of requested blocks are in the sequences whose sizes are larger than 120. Though DULO can increase the sizes of short sequences a little bit, and hence reduce execution time by 4.4% (up to 8.0%), its influence is limited. For the almost-all-random workload *tpc-h*, apparently DULO cannot create sequential disk requests from the application requests consisting of pure random blocks. So we see almost no improve-

Application	Num of block accesses (M)	Aggregate file size(MB)	Num of files	sequential refs
viewperf	0.3	495	289	99%
tpc-h	13.5	1187	49	3%
glimpse	3.1	669	43649	74%
multi1 (<i>cscope+gcc+viewperf</i>)	1.6	792	12514	75%
multi2 (<i>glimpse+tpc-h</i>)	16.6	1855	43696	16%

Table 2: Characteristics of the traces used in the simulations

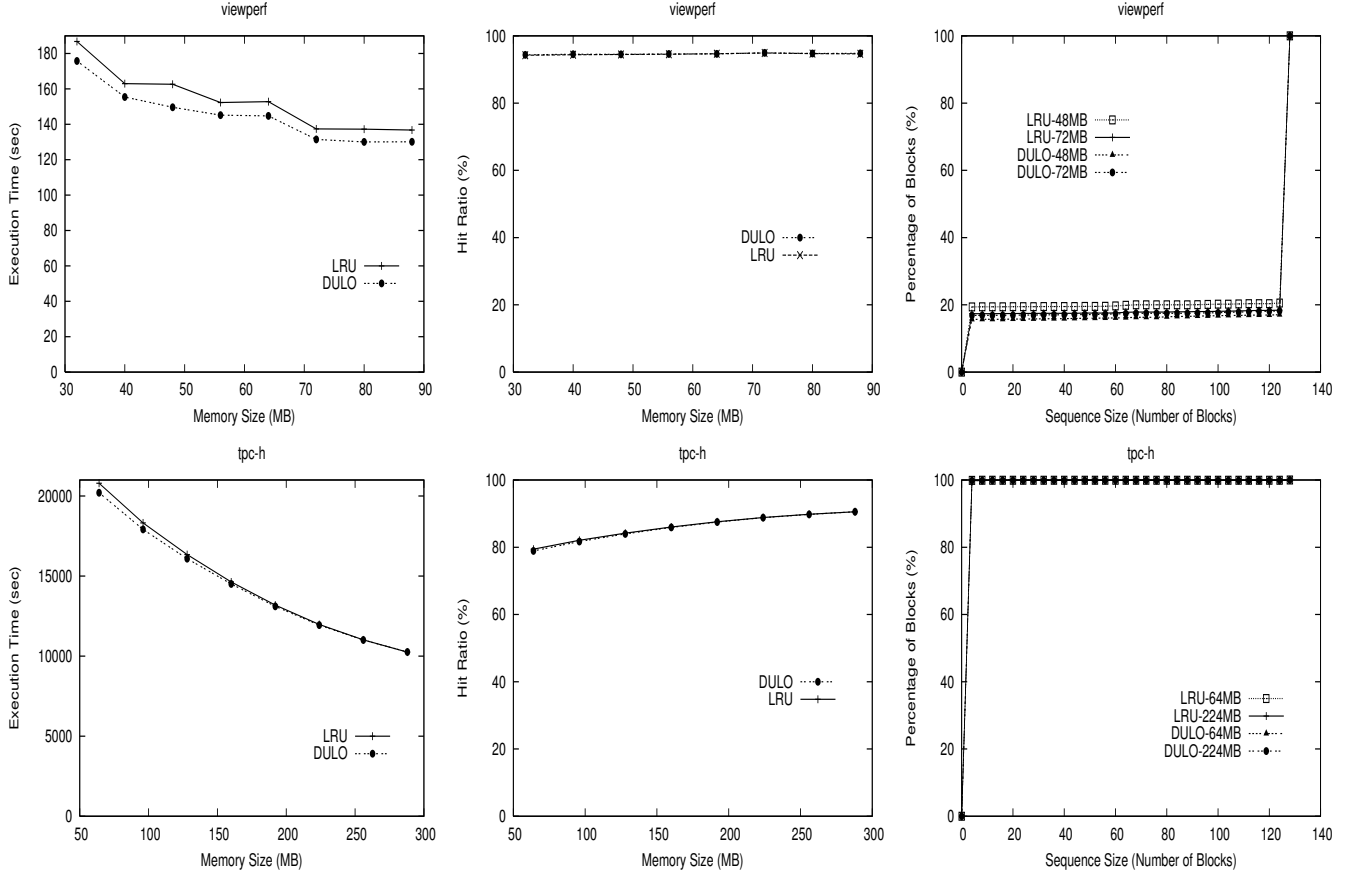


Figure 3: Execution times, hit ratios, and disk access sequence size distributions (CDF curves) of LRU caching and DULO caching schemes for *viewperf* with sequential request pattern and *tpc-h* with random request pattern.

ments from DULO.

DULO achieves substantial performance improvements for the workloads with mixed request patterns (see Figure 4). There are several observations from the figures. First, the sequence size increases are directly correlated to the execution time and hit ratio improvements. Let us take *multi1* as an example, with the memory size of 80MB, DULO makes 16.1% requested blocks appear in the sequences whose sizes are larger than 40 compared with 13.7% for LRU. Accordingly, there is an 8.5% execution time reduction and a 3.8% hit ratio increase. By contrast, with the memory size of 160MB, DULO makes 24.9% requested blocks appear in the sequences whose sizes are larger than 40 compared with 14.0% for LRU. Accordingly, there is a 55.3% execution time reduction and a 29.5% hit ratio increase. The corre-

lation clearly indicates that requested sequence size is a critical factor affecting disk performance and DULO makes its contributions through increasing the sequence size. DULO can increase the hit ratio by making prefetching more effective with long sequences and generating more hits on the prefetched blocks. Second, the sequential accesses are important for DULO to leverage the buffer cache filtering effect. We see that DULO does a better job for *glimpse* and *multi1* than for *multi2*. We know that *glimpse* and *multi1* have 74% and 75% of sequential accesses, while *multi2* has only 16% sequential accesses. The small portion of sequential accesses in *multi2* make DULO less capable to keep random blocks from being replaced because there are not sufficient sequentially accessed blocks to be replaced first. Third, *multi1* has more pronounced performance improve-

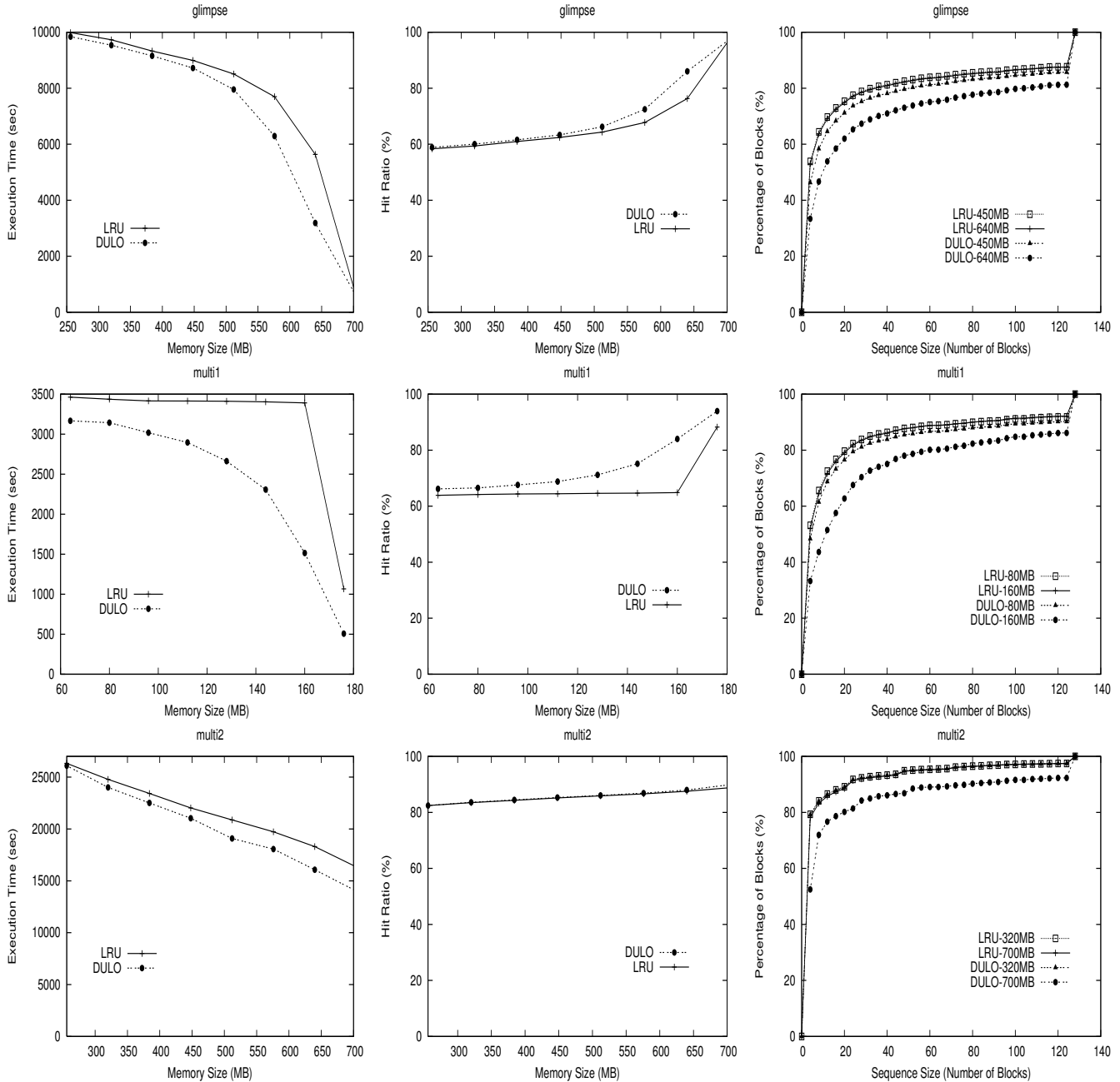


Figure 4: Execution times, hit ratios, and disk access sequence size distributions (CDF curves) of LRU caching and DULO caching schemes for *glimpse*, *multi1*, and *multi2* with mixed request patterns.

ments from DULO than *glimpse* does. This difference is mainly because DULO incidentally improves the LRU hit ratios by better exploiting temporal locality with the looping access pattern, for which LRU has well known inability (see e.g. [17]). By contrast, in the case of *multi2*, DULO can hardly improve its hit ratios, but is able to considerably reduce its execution times, which clearly demonstrates its effectiveness at exploiting spatial locality.

4.1.3 Parameter Sensitivity and Overhead Study

There are two parameters in the DULO scheme, the (maximum) sequencing bank size and the (minimal) evicting size. Both of these sizes should be related to the workload access patterns rather than memory size, because they are used to manage the sequentiality of accessed blocks. We use four workloads for the study, excluding *viewperf* because its memory demand is very small.

Table 3 shows the execution times change with varying

Bank Size (MB)	<i>tpc-h</i> (128M)	<i>glimpse</i> (640M)	<i>multi1</i> (160M)	<i>multi2</i> (640M)
1	16325	3652	1732	18197
2	16115	3611	1703	17896
4	15522	3392	1465	16744
8	15698	3392	1483	16737
16	15853	3412	1502	17001
32	15954	3452	1542	17226

Table 3: The execution times (seconds) with varying bank sizes (MB). Evicting section size is 64MB. Memory sizes are shown with their respective workload names.

Evicting Section Size (MB)	<i>tpc-h</i> (128M)	<i>glimpse</i> (640M)	<i>multi1</i> (160M)	<i>multi2</i> (640M)
32	15750	3852	1613	18716
64	15698	3392	1483	16737
128	-	3382	1406	16685
192	-	3361	-	16665
256	-	3312	-	16540
320	-	3342	-	16538

Table 4: The execution times (seconds) with varying evicting section sizes (MB). Sequencing bank size is 8MB. Memory sizes are shown with their respective workload names.

sequencing bank sizes. We observe that across the workloads with various access patterns, there is an optimal bank size roughly in the range from 4MB to 16MB. This is because a bank with too small size has little chance to form long sequences. Meanwhile, a bank size must be less than the evicting section size. When the bank size approaches the section size, the performance will degrade. This is because a large bank size causes the evicting section to be refilled too late and causes the random blocks in it to have to be evicted. So in our experiments we choose 8MB as the bank size.

Table 4 shows the execution times change with varying evicting section sizes. Obviously the larger the section size, the more control DULO will have on the eviction order of the blocks in it, which usually means better performance. The figure does show the trend. Meanwhile, the figure also shows that the benefits from the increased evicting section size saturate once the size exceeds the range from 64MB to 128MB. In our experiments, we choose 64MB as the section size because the memory demands of our workloads are relatively small.

The space overhead of DULO is its block table, whose size growth corresponds to the number of compulsory misses. Only a burst of compulsory misses could cause the table size to be quickly increased. However, the table space can be reclaimed by the system in a grace manner as described in Section 3.2. The time overhead of DULO is trivial because it is associated with the misses. Our simulations show that a miss is associated with one block sequencing operation including placing the block into the block table and comparing with its adjacent blocks, and 1.7 merge sort com-

parison operation in average.

4.2 The DULO Implementation

To demonstrate the performance improvements of DULO for applications running on a modern operating system, we implement DULO in the recent Linux kernel 2.6.11. One of the unique benefits from real system evaluation over trace simulation is that it can take all the memory usages into account, including process memory and file-mapped memory pages. For example, due to time and space cost constraints, it is almost impossible to faithfully record all the memory page accesses as a trace. Thus, the traces we used in the simulation experiments only reflect the file access activities through system calls. To present a comprehensive evaluation of DULO, our kernel implementation and system measurements effectively complement our trace simulation results.

Let us start with a brief description of the implementation of the Linux replacement policy, an LRU variant.

4.2.1 Linux Caching

Linux adopts an LRU variant similar to the 2Q replacement [16]. The Linux 2.6 kernel groups all the process pages and file pages into two LRU lists called the *active list* and the *inactive list*. As their names indicate, the active list is used to store recently actively accessed pages, and the inactive list is used to store those pages that have not been accessed for some time. A faulted-in page is placed at the head of the inactive list. The replacement page is always selected at the tail of the inactive list. An inactive page is promoted into the active list when it is accessed as a file page (by *mark_page_accessed()*), or it is accessed as a process page and its reference is detected at the inactive list tail. An active page is demoted to the inactive list if it is determined to have not been recently accessed (by *refill_inactive_zone()*).

4.2.2 Implementation Issues

In our prototype implementation of DULO, we do not replace the original Linux page frame reclaiming code with a faithful DULO scheme implementation. Instead, we opt to keep the existing data structure and policies mostly unchanged, and seamlessly adapt DULO into them. We make this choice, which has to tolerate some compromises of the original DULO design, to serve the purpose of demonstrating what improvements a dual locality consideration could bring to an existing spatial-locality-unaware system without changing its basic underlying replacement policy.

In Linux, we partition the inactive list into a staging section and an evicting section because the list is the place where new blocks are added and old blocks are replaced. Two LRU lists used in Linux instead of one assumed in the DULO scheme challenge the legitimacy of forming a sequence by using one bank in the staging section. We know

that the sequencing bank in DULO is intended to collect continuously accessed pages and form sequences from them, so that the pages in a sequence can be expected to be requested together and be fetched sequentially. With two lists, both newly accessed pages and not recently accessed active pages demoted from the active list might be added into the inactive list and probably be sequenced in the same bank⁴. Hence, two spatially sequential but temporally remote pages can possibly be grouped into one sequence, which is apparently in conflict with the sequence definition described at the beginning of Section 4. We address this issue by marking the demoted active pages and sequencing each type of page separately. Obviously, the Linux two-list structure provides fewer opportunities for DULO to identify sequences than those in one stack case where any hit pages are available for a possible sequencing with faulted-in pages.

The anonymous pages that do not yet have mappings on disk are treated as random blocks until they are swapped out and are associated with some disk locations. To map the LBN (Logical Block Number) of a block into a one-dimensional physical disk address, we use the technique described in [33] to extract track boundaries. To characterize accurately block location sequentiality, all the defective and spare blocks on disk are counted. We also artificially place a dummy block between the blocks on a track boundary in the mapping to show the two blocks are non-sequential.

There are two types of I/O operations, namely file access and VM paging. In the experiments, we set the sequencing bank size as 8MB, and the evicting section size as 64MB, the same as those adopted in the simulations.

4.2.3 Case Study I: File Accesses

In the first case we study the influence of the DULO scheme on file access performance. For this purpose, we installed a Web server running a general hypertext cross-referencing tool — Linux Cross-Reference (LXR) [24]. This tool is widely used by Linux developers for searching Linux source code. The machine we use is a Gateway desktop with Intel P4 1.7GHz processor, a 512MB memory, and Western Digital WD400BB hard disk of 7200 RPM. The OS is SuSE Linux 9.2 with the Linux 2.6.11 kernel. The file system is Ext2. We use LXR 0.3 search engine on the Apache 2.0.50 HTTP Server, and use Glimpse 4.17.3 as the freetext search engine. The file set for the searching is Linux 2.6.11.9 source code, whose size is 236MB. Glimpse divides the files into 256 partitions, indexes the file set based on partitions, and generates a 12MB index file showing the keyword locations in terms of partitions. To serve a search query, glimpse searches the index file first, then accesses the files included in the partitions matched in the index files. On the client side, we used WebStone 2.5 [36] to generate 25 clients concurrently submitting freetext search queries. Each client randomly picks up a keyword from a pool of 50 keywords and

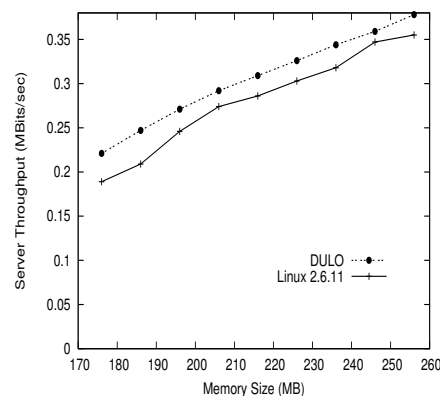


Figure 5: Throughputs of LXR search on the original Linux kernel and DULO instrumented kernel with varying memory sizes.

sends it to the server. It sends its next query request once it receives the results of its previous query. We randomly select 25 Linux symbols from file */boot/System.map* and another 25 popular OS terms such as “*lru*”, “*scheduling*”, “*page*” as the pool of candidate query keywords. Each experiment lasts for 15 minutes. One client always uses the same sequence of keyword queries in each experiment. The metric we use is the query system throughput represented by MBits/sec, which means the number of Mega bits of query results returned by the server per second. This metric is also used for reporting WebStone benchmark results. Because in the experiments the query processing is I/O-intensive, the metric is suitable to measure the effectiveness of the memory and disk systems.

Figure 5 shows the server throughputs for the original Linux 2.6.11 kernel and its DULO instrumented counterpart with various memory sizes. The reported memory sizes are those available for the kernel and user applications. We adopt relatively small memory sizes because of the limited size of the file set for search. The figure shows that DULO helps improve the benchmark throughput by 5.1% to 16.9%, and the trend also holds for the hit ratio curves (not shown in this paper). To understand the performance improvements, we examine the disk layout of the glimpse partitions (i.e., the sets of files the glimpse searches for a specific keyword). There are a small percentage of files in a partition that are located non-continuously with the rest of its files. The fact that a partition is the glimpse access unit makes accesses to those files be random accesses interleaved in the sequential accesses. DULO identifies these isolated files and keeps them in memory with priority. Then the partition can be scanned without abruptly moving the disk head, even if the partition contains isolated small files. To prepare the aforementioned experiments, we *untar* the compressed kernel 2.6.11.9 on the disk with 10% of its capacity occupied. To verify our performance explanation, we manually copy the source code files to an unoccupied disk and make all the files in a glimpse partition be closely allocated on the disk. Then we repeat the experiments. This time, we see little difference between the

DULO instrumented kernel and the original kernel, which clearly shows that (1) DULO can effectively and flexibly exploit spatial locality without carefully tuning system components, which is sometimes infeasible; (2) The additional running overhead introduced by DULO is very small.

4.2.4 Case Study II: VM Paging

In the second case we study the influence of the DULO scheme on VM paging performance. For this purpose, we use a typical scientific computing benchmark — sparse matrix multiplication (SMM) from an NIST benchmark suite SciMark2 [31]. The system settings are the same as those adopted in the previous case study. The SMM benchmark multiplies a sparse matrix with a vector. The matrix is of size $N \times N$, and has M non-zero data regularly dispersed in its data geometry, while the vector has a size of N ($N = 2^{20}$ and $M = 2^{23}$). The data type is 8Byte *double*. In the multiplication algorithm, the matrix is stored in a compressed-row format so that all the non-zero elements are continuously placed in a one-dimensional array with two index arrays recording their original locations in the matrix. The total working set, including the result vector and the index arrays, is around 116MB. To cause the system paging and stress the swap space accesses, we have to adopt small memory sizes, from 90MB to 170MB, including the memory used by the kernel and applications. The benchmark is designed to repeat the multiplication computation 15 times to collect data.

To increase spatial locality of swapped-out pages in disk swap space, Linux tries to allocate continuous swap slots on disk to sequentially reclaimed anonymous pages in the hope that they would be swapped-in in the same order efficiently. However, the data access pattern in SMM foils the system effort. In the program, SMM first initializes the arrays one by one. This thereafter causes each array to be swapped out continuously and be allocated on the disk sequentially when the memory cannot hold the working set. However, in the computation stage, the elements that are accessed in the vector array are determined by the matrix locations of the elements in the matrix array. Thus, those elements are irregularly accessed, but they are continuously located on the disk. The swap-in accesses to the vector arrays turn into random accesses, while the elements of matrix elements are still sequentially accessed. This explains the SMM execution time differences between on the original kernel and on DULO instrumented kernel (see Figure 6). DULO significantly reduces the execution times by up to 43.7%, which happens when the memory size is 135MB. This is because DULO detects the random pages in the vector array and caches them with priority. Because the matrix is a sparse one, the vector array cannot obtain sufficiently frequent reuses to allow the original kernel to keep them from being paged out. Furthermore, the similar execution times between the two kernels when there is enough memory to hold the working set shown

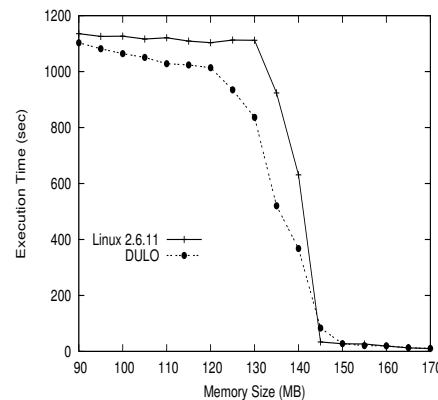


Figure 6: SMM execution times on the original Linux kernel and DULO instrumented kernel with varying memory sizes.

in the figure suggest that the DULO overhead is small.

5 Related Work

Because of the serious disk performance bottleneck that has existed over decades, many researchers have attempted to avoid, overlap, or coordinate disk operations. In addition, there are studies on the interactions of these techniques and on their integration in a cooperative fashion. Most of the techniques are based on the existence of locality in disk access patterns, either temporal locality or spatial locality.

5.1 Buffer caching

One of the most actively researched area on improving I/O performance is buffer caching, which relies on intelligent replacement algorithms to identify and keep active pages in memory so that they can be re-accessed without actually accessing the disk later. Over the years, numerous replacement algorithms have been proposed. The oldest and yet still widely adopted algorithm is the Least Recently Used (LRU) algorithm. The popularity of LRU comes from its simple and effective exploitation of temporal locality: a block that is accessed recently is likely to be accessed again in the near future. There are also a large number of other algorithms proposed such as 2Q [16], MQ [38], ARC [25], LIRS [17] et al. All these algorithms focus only on how to better utilize temporal locality, so that they are able to better predict the blocks to be accessed and try to minimize page fault rate. None of these algorithms considers spatial locality, i.e., how the stream of faulted pages is related to their disk locations. Because of the non-uniform access characteristic of disks, the distribution of the pages on disk can be more performance-critical than the number of the pages itself. In other words, the *quality* of the missed pages deserves at least the same amount of attention as their *quantity*. Our DULO scheme introduces spatial locality into the consideration of page replacement and thus makes replacement algorithms aware of

page placements on the disk.

5.2 I/O Prefetching

Prefetching is another actively researched area on improving I/O performance. Modern operating systems usually employ sophisticated heuristics to detect sequential block accesses so as to activate prefetching, as well as adaptively adjust the number of blocks to be prefetched within the scope of one individual file [5, 28]. System-wide file access history has been used in probability-based predicting algorithms, which track sequences of file access events and evaluate probability of file occurring in the sequences [11, 19, 20, 23]. This approach can perform prefetching across files and achieve a high prediction accuracy due to its use of historical information.

The performance advantages of prefetching coincide with sequential block requests. While prefetchers by themselves cannot change the I/O request stream in any way as the buffer cache does, they can take advantage of the more sequential I/O request streams resulted from the DULO cache manager. In this sense, DULO is a complementary technique to prefetching. With the current intelligent prefetching policies, the efforts of DULO on sequential accesses can be easily translated into higher disk performance.

5.3 Integration between Caching and Prefetching

Many research papers on the integration of caching and prefetching consider the issues such as the allocations of memory to cached and prefetched blocks, the aggressiveness of prefetching, and use of application-disclosed hints in the integration [2, 6, 12, 21, 7, 22, 27, 35]. Sharing the same weakness as those in current caching policies, this research only utilizes the temporal locality of the cached/prefetched blocks and uses hit ratio as metric in deciding memory allocations. Recent research has found that prefetching can have a significant impact on caching performance, and points out that the number of aggregated disk I/Os is a much more accurate indicator of the performance seen by applications than hit ratio [5].

Most of the proposed integration schemes rely on application-level hints about I/O access patterns provided by users [6, 7, 22, 27, 35]. This reliance certainly limits their application scope, because users may not be aware of the patterns or source code may not be available. The work described in [21, 12] does not require additional user support and thus is more related to our DULO design.

In paper [21], a prefetching scheme called *recency-local* is proposed and evaluated using simulations. *Recency-local* prefetches the pages that are nearby the one being referenced in the LRU stack⁵. It takes a reasonable assumption — pages adjacent to the one being demanded in the LRU stack would

likely be used soon, because it is likely that the same access sequence would be repeated. The problem is that those nearby pages in the LRU stack may not be adjacent to the page being accessed on disk (i.e., sharing spatial locality). In fact, this is the scenario that is highly likely to happen in a multi-process environment, where multiple processes that access different files interleavingly feed their blocks into the common LRU stack. Prefetching requests involving disk seeks make little sense to improving I/O performance, and can hurt the performance due to possible wrong predictions. If we re-order the blocks in a segment of an LRU stack according to their disk locations, so that adjacent blocks in the LRU stack are also close to each other on disk, then replacing and prefetching of the blocks can be conducted in a spatial locality conscious way. This is one of the motivations of DULO.

Another recent work is described in paper [12], in which cache space is dynamically partitioned among sequential blocks, which have been prefetched sequentially into the cache, and random blocks, which have been fetched individually on demand. Then a marginal utility is used to compare constantly the contributions to the hit rate between the allocation of memory to sequential blocks and that to random blocks. More memory is allocated to the type of blocks that can generate a higher hit rate, so that the system-wide hit rate is improved. However, a key observation is unfortunately ignored here, i.e., sequential blocks can be brought into the cache much faster than an equivalent number of random blocks due to their spatial locality. Therefore, the misses of random blocks should count more in their contribution to final performance. In their marginal utility estimations, misses on the two types of blocks are equally treated without giving preference to random blocks, even though the cost of fetching random blocks is much higher. Our DULO gives random blocks more weight for being kept in cache to compensate for their high fetching cost.

While modern operating systems do not support caching and prefetching integration designs yet, we do not pursue in this aspect in our DULO scheme in this paper. We believe that introducing dual locality in these integration schemes will certainly improve their performance, and that it remains as our future work to investigate the amount of its benefits.

5.4 Other Related Work

Because disk head seek time far dominates I/O data transfer time, to effectively utilize the available disk bandwidth, there are techniques to control the data placement on disk [1, 3] or reorganize selected disk blocks [14], so that related objects are clustered and the accesses to them become more sequential. In DULO, we take an alternative approach in which we try to avoid random small accesses by preferentially keeping these blocks in cache and thereby making more accesses sequential. In comparison, our approach is capable of adapting

itself to changing I/O patterns and is a more widely applicable alternative to the disk layout control approach.

Finally, we point out some interesting work analogous to our approach in spirit. [10] considers the difference in performance across heterogeneous storage devices in storage-aware file buffer replacement algorithms, which explicitly give those blocks from slow devices higher weight to stay in cache. To do so, the algorithms can adjust the stream of block requests to have more fast device requests by filtering slow device requests to improve caching efficiency. In papers [29, 39, 40], the authors propose to adapt replacement algorithms or prefetching strategies to influence the I/O request streams for disk energy saving. With the customized cache filtering effect, the I/O stream to disks becomes more bursty separated by long idle time to increase disk power-down opportunities in the single disk case, or becomes more unbalanced among the requests' destination disks to allow some disks to have longer idle times to power down. All this work leverages the cache's buffering and filtering effects to influence I/O access streams and to make them friendly to specific performance characteristics of disks for their respective objectives, which is the philosophy shared by our DULO. The uniqueness of DULO is that it influences disk access streams to make them more sequential to reduce disk head seeks.

6 Limitations of our Work

While the DULO scheme exhibits impressive performance improvements in average disk latency and bandwidth, as well as the application runtimes, there are some limitations worth pointing out. First, though DULO attempts to provide random blocks with a caching privilege to avoid the expensive I/O operations on them, there is little that DULO can do to help I/O requests incurred by compulsory misses or misses to random blocks that have not been accessed for a long time. In addition, the temporal-locality-only caching policy is able to cache frequently accessed random blocks, and there is no need for DULO's help. This discussion also applies to those short sequences whose sizes cannot well amortize the disk seeking cost. Second, we present our DULO scheme based on the LRU stack. For implementation purposes, we adapt it to the 2Q-like Linux replacement policy. The studies of how to adapt DULO on other advanced caching algorithms and understanding the interaction between DULO and the characteristics of each algorithm are necessary and in our research plan. Third, as we have mentioned, it remains as our future work to study the integration between caching and prefetching policies in the DULO scheme.

7 Conclusions

In this paper, we identify a serious weakness of lacking spatial locality exploitation in I/O caching, and propose a new

and effective memory management scheme, DULO, which can significantly improve I/O performance by exploiting both temporal and spatial locality. Our experiment results show that DULO can effectively reorganize application I/O request streams mixed with random and sequential accesses in order to provide a more disk-friendly request stream with high sequentiality of block accesses. We present an effective DULO replacement algorithm to carefully tradeoff random accesses with sequential accesses and evaluate it using traces representing representative access patterns. Besides extensive simulations, we have also implemented DULO in a recent Linux kernel. The results of performance evaluation on both buffer cache and virtual memory system show that DULO can significantly improve the performance up to 43.7%.

Acknowledgment

We are grateful to Ali R. Butt, Chris Gniady, and Y. Charlie Hu at Purdue University for providing us with their file I/O traces. We are also grateful to the anonymous reviewers who helped improve the quality of the paper. We thank our colleagues Bill Bynum, Kei Davis, and Fabrizio Petrini to read the paper and their constructive comments. The research was supported by Los Alamos National Laboratory under grant LDRD ER 20040480ER, and partially supported by the National Science Foundation under grants CNS-0098055, CCF-0129883, and CNS-0405909.

References

- [1] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. Nugent, F. I. Popovici, "Transforming Policies into Mechanisms with In-fokernel", *Proc. of SOSP '03*, October 2003.
- [2] S. Albers and M. Buttner, "Integrated prefetching and caching in single and parallel disk systems", *Proc. of SPAA '03*, June, 2003.
- [3] D. Black, J. Carter, G. Feinberg, R. MacDonald, S. Mangalat, E. Sheinbrood, J. Sciver, and P. Wang, "OSF/1 Virtual Memory Improvements", *Proc. of the USENIX Mac Symposium*, November 1991.
- [4] J. Bucy and G. Ganger, "The DiskSim Simulation Environment Version 3.0 Reference Manual", *Technical Report CMU-CS-03-102*, Carnegie Mellon University, January 2003.
- [5] A. R. Butt, C. Gniady, and Y. C. Hu, "The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms", *Proc. of SIGMETRICS '05*, June, 2005.
- [6] P. Cao, E. W. Felten, A. Karlin and K. Li, "A Study of Integrated Prefetching and Caching Strategies", *Proc. of SIGMETRICS '95*, May 1995.
- [7] P. Cao, E. W. Felten, A. Karlin and K. Li, "Implementation and Performance of Integrated Application-Controlled

- Caching, Prefetching and Disk Scheduling”, *ACM Transaction on Computer Systems*, November 1996.
- [8] P. Cao, S. Irani, “Cost-Aware WWW Proxy Caching Algorithms”, *Proc. of USENIX '97*, December, 1997.
 - [9] P. Cao, E. W. Felten and K. Li, “Application-Controlled File Caching Policies”, *Proc. of USENIX Summer '94*, 1994.
 - [10] B. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Storage-Aware Caching: Revisiting Caching For Heterogeneous Storage Systems”, *Proc. of FAST '02*, January 2002.
 - [11] J. Griffioen and R. Appleton, “Reducing file system latency using a predictive approach”, *Proc. of USENIX Summer '94*, June 1994.
 - [12] B. Gill and D. S. Modha, “SARC: Sequential Prefetching in Adaptive Replacement Cache,” *Proc. of USENIX '05*, April, 2005.
 - [13] M. Gorman, “Understanding the Linux Virtual Memory Manager”, *Prentice Hall*, April, 2004.
 - [14] W. W. Hsu, H. C. Young and A. J. Smith, “The Automatic Improvement of Locality in Storage Systems”, *Technical Report CSD-03-1264, UC Berkeley*, July, 2003.
 - [15] S. Iyer and P. Druschel, “Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O”, *Proc. of SOSP '01*, October 2001.
 - [16] T. Johnson and D. Shasha, “2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm”, *Proc. of VLDB '94*, 1994, pp. 439-450.
 - [17] S. Jiang and X. Zhang, “LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance”, *Proc. of SIGMETRICS '02*, June 2002.
 - [18] S. Jiang, F. Chen and X. Zhang, “CLOCK-Pro: An Effective Improvement of the CLOCK Replacement”, *Proc. of USENIX '05*, April 2005.
 - [19] T. M. Kroege and D.D.E. Long, “Predicting file-system actions from prior events”, *Proc. of USENIX Winter '96*, January 1996.
 - [20] T. M. Kroege and D.D.E. Long, “Design and implementation of a predictive file prefetching algorithm”, *Proc. of USENIX '01*, January 2001.
 - [21] S. F. Kaplan, L. A. McGeoch and M. F. Cole, “Adaptive Caching for Demand Prepaging”, *Proc. of the International Symposium on Memory Management*, June, 2002.
 - [22] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. Felton, G. Gibson, A. R. Karlin and K. Li, “A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching”, *Proc. of OSDI '96*, 1996.
 - [23] H. Lei and D. Duchamp, “An Analytical Approach to File Prefetching”, *Proc. USENIX '97*, January 1997.
 - [24] Linux Cross-Reference, URL : <http://lxr.linux.no/>.
 - [25] N. Megiddo, D. Modha, “ARC: A Self-Tuning, Low Overhead Replacement Cache”, *Proc. of FAST '03*, March 2003, pp. 115-130.
 - [26] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering”, *Proc. of SIGMOD '93*, 1993.
 - [27] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky and J. Zelenka, “Informed Prefetching and Caching”, *Proc. of SOSP '95*, 1995.
 - [28] R. Pai, B. Pulavarty and M. Cao, “Linux 2.6 Performance Improvement through Readahead Optimization”, *Proceedings of the Linux Symposium*, July 2004.
 - [29] A. E. Papathanasiou and M. L. Scott, “Energy Efficient Prefetching and Caching”, *Proc. of USENIX '04*, June, 2004.
 - [30] R. Love, “Linux Kernel Development (2nd Edition)”, *Novell Press*, January, 2005.
 - [31] SciMark 2.0 benchmark, URL: <http://math.nist.gov/scimark2/>
 - [32] A. J. Smith, “Sequentiality and Prefetching in Database Systems”, *ACM Trans. on Database Systems*, Vol. 3, No. 3, 1978.
 - [33] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger, “Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics”, *Proc. of FAST '02*, January, 2002.
 - [34] E. Shriver, C. Small, K. A. Smith, “Why Does File System Prefetching Work?”, *Proc. of USENIX '99*, June, 1999.
 - [35] A. Tomkins, R. H. Patterson and G. Gibson, “Informed Multi-Process Prefetching and Caching”, *Proc. of SIGMETRICS '97*, June, 1997.
 - [36] WebStone — The Benchmark for Web Servers, URL : <http://www.mindcraft.com/benchmarks/webstone/>
 - [37] N. Young, “Online file caching”, *Proc. of SODA '98*, 1998.
 - [38] Y. Zhou, Z. Chen and K. Li. “Second-Level Buffer Cache Management”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 7, July, 2004.
 - [39] Q. Zhu, F. M. David, C. F. Devaraj, Z. Li, Y. Zhou, and P. Cao, “Reducing Energy Consumption of Disk Storage Using Power-Aware Cache Management”, *Proc. of HPCA '04*, February 2004.
 - [40] Q. Zhu, A. Shankar and Y. Zhou, “PB-LRU: A Self-Tuning Power Aware Storage Cache Replacement Algorithm for Conserving Disk Energy”, *Proc. of ICS '04*, June, 2004

Notes

¹We use *page* to denote a memory access unit, and *block* to denote a disk access unit. They can be of different sizes. For example, a typical Linux configuration has a 4KB page and a 1KB block. A page is then composed of one or multiple blocks if it has a disk mapping.

²With a seek reduction disk scheduler, the actual seek time between consecutive accesses should be less than the average time. However, this does not affect the legitimacy of the discussions in the section as well as its conclusions.

³The definition of sequence can be easily extended to a cluster of blocks whose disk locations are close to each other and can be used to amortize the cost of one disk operation. We limit the concept to being strictly sequential in this paper because that is the dominant case in practice.

⁴This issue might not arise if the last timestamps of these two types of blocks cannot meet the sequencing criteria listed in section 3.3, but there is no guarantee of this.

⁵LRU stack is the data structure used in the LRU replacement algorithm. The block ordering in it reflects the order of block accesses.

Second-Tier Cache Management Using Write Hints

Xuhui Li
University of Waterloo

Ashraf Aboulmaga
University of Waterloo

Kenneth Salem
University of Waterloo

Aamer Sachedina
IBM Toronto Lab

Shaobo Gao
University of Waterloo

Abstract

Storage servers, as well as storage clients, typically have large memories in which they cache data blocks. This creates a two-tier cache hierarchy in which the presence of a first-tier cache (at the storage client) makes it more difficult to manage the second-tier cache (at the storage server). Many techniques have been proposed for improving the management of second-tier caches, but none of these techniques use the information that is provided by *writes* of data blocks from the first tier to help manage the second-tier cache. In this paper, we illustrate how the information contained in writes from the first tier can be used to improve the performance of the second-tier cache. In particular, we argue that there are very different reasons why storage clients write data blocks to storage servers (e.g., cleaning dirty blocks vs. limiting the time to recover from failure). These different types of writes can provide strong indications about the current state and future access patterns of a first-tier cache, which can help in managing the second-tier cache. We propose that storage clients inform the storage servers about the types of writes that they perform by passing *write hints*. These write hints can then be used by the server to manage the second-tier cache. We focus on the common and important case in which the storage client is a database system running a transactional (OLTP) workload. We describe, for this case, the different types of write hints that can be passed to the storage server, and we present several cache management policies that rely on these write hints. We demonstrate using trace driven simulations that these simple and inexpensive write hints can significantly improve the performance of the second-tier cache.

1 Introduction

Current storage servers have large memories which they use to cache data blocks that they serve to their clients. The storage clients, in turn, typically cache these data

blocks in their own memories. This creates a *two-tier cache hierarchy* in which both the storage server and the storage client cache the same data with the goal of improving performance.¹

Managing the *second-tier* (storage server) cache is more difficult than managing the *first-tier* (storage client) cache for several reasons. One reason is that the first-tier cache captures the accesses to the hot blocks in the workload. This reduces the temporal locality in the accesses to the second-tier cache, which makes recency-based replacement policies (e.g., LRU or Clock) less effective for the second tier.

Another reason why managing second-tier caches is difficult is that the second-tier cache may include blocks that are already present in the first-tier cache. Accesses to these blocks would hit in the first tier, so caching them in the second tier is a poor use of available cache space. Hence, second-tier cache management has the additional requirement of trying to maintain *exclusiveness* between the blocks in the first and second tiers [20].

Managing second-tier caches is also difficult because the cache manager needs to make placement and replacement decisions without full knowledge of the access pattern or cache management policy at the first tier. For example, a request to the second-tier for a block indicates a first-tier miss on that block, but does not provide information on how many first-tier hits to the block preceded this miss.

The difficulty of managing second-tier caches has been recognized in the literature, and various techniques for second-tier cache management have been proposed. Examples of these techniques include:

- Using cache replacement policies that rely on frequency as well as recency to manage second-tier caches [22].
- Passing hints from the storage client to the storage server about which requested blocks are likely to be retained and which are likely to be evicted [8, 5].

- Using knowledge of the algorithms and access patterns of the storage client to prefetch blocks into the second-tier cache [18, 2].
- Placing blocks into the second-tier cache not when they are *referenced* but when they are *evicted* by the first-tier cache [20, 6, 21].
- Evicting blocks requested by the first tier quickly from the second-tier cache [8, 20, 2].
- Using a single cache manager to manage both the client and the server caches [11].

Some of these techniques place extra responsibilities on the storage client for managing the storage server cache, and therefore require modifying the storage client [8, 20, 11, 5]. Other techniques do not require any modifications to the storage client, but spend CPU and I/O bandwidth trying to infer the contents of the storage client cache and predict its access patterns [1, 6, 2]. A common characteristic of all these techniques is that they do not have any special treatment for *writes of data blocks* from the storage client to the storage server.

In this paper, we focus on using *write requests* from the storage client to improve the performance of the storage server cache. Storage clients write data blocks to the storage server for different reasons. For example, one reason is writing a dirty (i.e., modified) block while evicting it to make room in the cache for another block. Another, very different, reason is periodically writing frequently modified blocks to guarantee reliability. The different types of writes provide strong indications about the state of the first-tier cache and the future access patterns of the storage client, and could therefore be used to improve cache management at the storage server.

We propose associating with every write request a *write hint* indicating its type (i.e., why the storage client is writing this block). We also present different methods for using these write hints to improve second-tier cache replacement, either by adding hint-awareness to existing replacement policies (e.g., MQ [21] and LRU) or by developing new hint-based replacement policies.

Our approach requires modifying the storage client to provide write hints. However, the necessary changes are simple and cheap. In particular, we are not requiring the storage client to make decisions about the management of the second-tier cache. We are only requiring the storage client to choose from a small number of explanations of why it is writing each block it writes, and to pass this information to the storage server as a write hint.

The write hints that we consider in this paper are fairly general, and could potentially be provided by a variety of storage clients. However, to explore the feasibility and efficacy of the proposed write hints, we focus on one

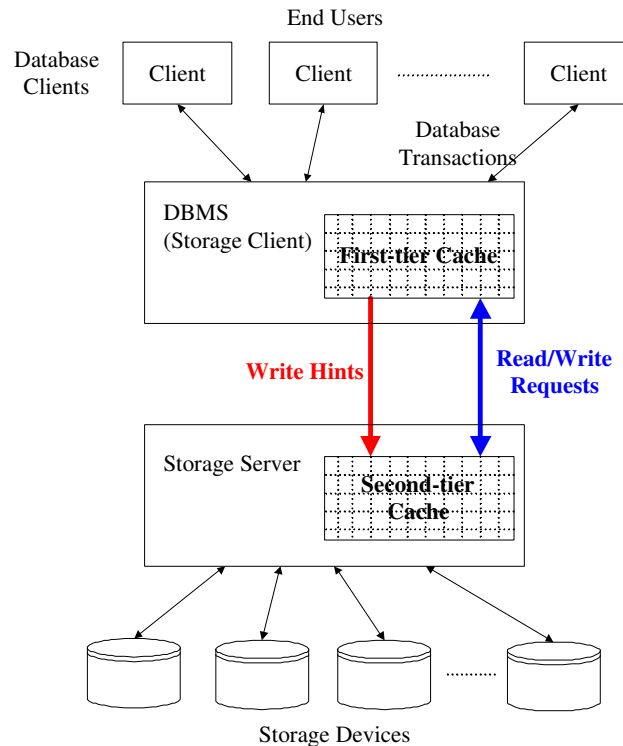


Figure 1: DBMS as the Storage Client

common and important scenario: a database management system (DBMS) running an on-line transaction processing (OLTP) workload as the storage client (Figure 1). For this scenario, we demonstrate using trace driven simulations that write hints can improve the performance of the recently proposed MQ cache replacement policy by almost 30%, and that TQ, a new hint-based replacement policy that we propose, can perform twice as well as MQ.

Our approach, while not transparent to the storage client, has the following key advantages:

- It is simple and cheap to implement at the storage server. There is no need to simulate or track the contents of the first-tier cache.
- It is purely opportunistic, and does not place additional load on the storage devices and network. When the storage server receives a write request, the request (a) contains a copy of the data to be written, and (b) must be flushed to the storage device at some point in time. Thus, if the second-tier cache manager decides, based on the write hints, to cache the block contained in a write request, it does not need to fetch this block from the storage device. On the other hand, if the second-tier cache manager decides not to cache the block contained in a write request, it has to flush this block to the storage de-

vice, but this flushing operation must be performed in any case, whether or not hints are used.

- As mentioned earlier, the first-tier cache typically captures most of the temporal locality in the workload. Thus, many reads will be served from the first-tier cache. Writes, on the other hand, must go to the second tier. Thus, the second-tier cache will see a higher fraction of writes in its workload than if it were the only cache in the system. This provides many opportunities for generating and using write hints.
- Using write hints is complementary to previous approaches for managing second-tier caches. We could exploit other kinds of hints, demotion information, or inferences about the state of the first-tier cache in addition to using the write hints.
- If the workload has few writes (e.g., a decision-support workload), the behavior of the proposed hint-aware replacement policies will degenerate to that of the underlying hint-oblivious policies. In that case, we expect neither benefit nor harm from using write hints.

Our contributions in this paper can be summarized as follows. We propose different types of write hints that can be generated by storage clients, and we propose second-tier cache replacement policies that exploit these hints. We evaluate the performance of these policies using traces collected from a real commercial DBMS running the industry standard TPC-C benchmark, and we compare them to the hint-oblivious alternatives. We also study an *optimal* replacement technique to provide an upper bound on how well we can do at the second tier.

The rest of this paper is organized as follows. In Section 2, we give some background about the architecture of a modern DBMS and its characteristics as a storage client. In Section 3, we present our proposal for using write hints, and in Section 4, we present three cache replacement policies that use these hints. Section 5 presents an evaluation of the proposed policies. Section 6 provides an overview of related work. We present our conclusions in Section 7.

2 Background

The I/O workload experienced by a storage server depends on the properties of its clients. Since we are considering a scenario in which the storage client is a DBMS, we first present, in this section, the relevant aspects of the process architecture and buffer management of a modern commercial DBMS. The specifics of this presentation are taken from DB2 Universal Database [9].

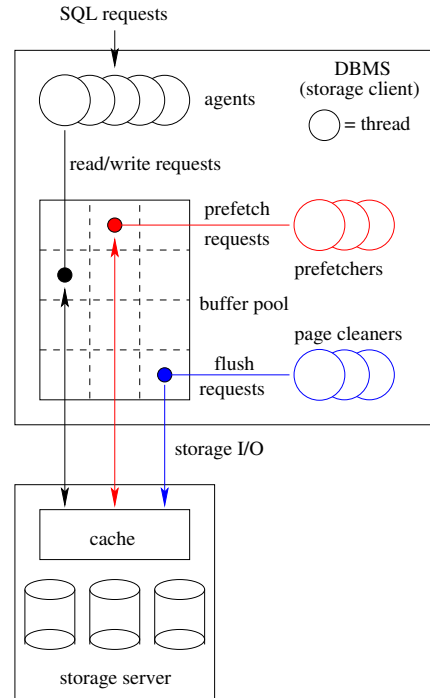


Figure 2: DBMS Architecture

However, similar features are found in other major commercial and open-source database management systems.

Figure 2 provides a simplified illustration of the multi-threaded (or multi-process, depending on the platform) execution architecture of the DBMS. The DBMS is capable of processing several application SQL requests concurrently. One or more threads, known as agents, are used to execute each SQL statement. As the agents run, they read and update the database structures, such as tables and indexes, through a block-oriented buffer pool. The DBMS may actually maintain several, independently managed buffer pools (not illustrated in Figure 2). Together, these pools constitute the storage client cache.

Each buffer pool is managed using a clock-based algorithm, so recency of reference is important in replacement decisions. However, the replacement policy also considers a number of other factors, including the type of data in the block, whether the block is clean or dirty, and the expected access pattern of the last agent to have used the block. Blocks are loaded into the buffer pool on demand from agents. Depending on the type of query being executed, prefetching may also be employed as a means of removing demand paging delays from the critical paths of the agents. Agents send read-ahead requests to a prefetching queue, which is serviced by a pool of prefetching threads. Prefetching threads retrieve blocks

from the underlying storage system and load them into the buffer pool, replacing blocks as necessary.

As agents run, they may modify the contents of blocks that are cached in the buffer pools. Modified (dirty) data blocks are generally not written immediately to the underlying storage system. Instead, one or more threads known as *page cleaners* are used to implement asynchronous (with respect to the agents) copy-back of dirty blocks from the buffer pool. In the event that the buffer replacement policy calls for the eviction of an updated block that has not been cleaned by a page cleaner, the agent (or prefetcher) that is responsible for the replacement flushes (writes) the dirty block back to the underlying storage system before completing the replacement. Note that flushing a dirty block does not by itself remove that block from the buffer pool. It simply ensures that the underlying storage device holds an up-to-date copy of the block.

The page cleaners must choose which dirty blocks to copy back to the storage system. There are two issues which affect this choice. First, the page cleaners try to ensure that blocks that are likely to be replaced by the agents will be clean at the time of the replacement. This removes the burden and latency associated with flushing dirty blocks from the execution path of the agents. To accomplish this, the page cleaners try to flush dirty blocks that would otherwise be good candidates for replacement.

The second issue considered by the page cleaners is failure recovery time. The DBMS uses write-ahead logging to ensure that committed database updates will survive DBMS failures. When the DBMS is recovering from a failure, the log is replayed to recreate any updates that were lost because they had not been flushed to the underlying storage system prior to the failure. The amount of log data that must be read and replayed to recover the proper database state depends on the age of the *oldest* changes that are in the buffer pool at the time of the failure. By copying relatively old updates from the buffer pools to the storage system, the page cleaners try to ensure that a configurable recovery time threshold will not be exceeded.

Several aspects of these mechanisms are worth noting. First, block writes to the underlying storage system usually do not correspond to evictions from the DBMS buffer pools. Writes correspond closely to evictions only when they are performed synchronously, by the agents. However, in a well-tuned system, the page cleaners try to ensure that such synchronous block writes are rare. Thus, if management of the storage server cache depends on knowledge of evictions from the client cache, that knowledge must be obtained by some other means, e.g., through the introduction of an explicit DEMOTE operation [20]. Second, the replacement algorithm used

to manage the DBMS buffer pool is complex and uses application-specific information. This poses a challenge to storage server cache managers that rely on simulation of the storage client as a means of predicting which blocks are in the client's cache [2].

3 Write Hints

As was noted in Section 1, we propose to use write requests to improve the performance of the storage server cache. Each write request generated by the storage client includes a copy of the block being written, so write requests provide low-overhead opportunities to place blocks into the storage server's cache. Furthermore, the fact that the storage client has written block *b* to the storage server may also provide some clues as to the state of the storage client's cache. The storage server can exploit these hints to improve the exclusiveness of its cache with respect to the client's.

What can the storage server infer about the storage client from the occurrence of a write? One key to answering this question is the fact that there are several distinct reasons why the storage client issues write requests, as described in Section 2. The first reason is block replacement: if the client wants to replace block *b* and *b* has been updated, then the client must write *b* back to the storage server before replacing it. We call such write requests *replacement writes*. The second reason for writing is to limit data loss in the event of a failure at the storage client. Thus, the storage client may write a block to the storage server even though that block is not a likely replacement candidate, in order to ensure the recoverability of changes that have been made to that block. We call such write requests *recoverability writes*.

A second key issue is the relationship between the time of the client's write of block *b* and the time of *b*'s eviction from the client's cache. In some cases, the client writes a dirty block *b* to the storage server because it is about to evict *b* from its cache. In the DBMS architecture described in Section 2, such writes may be generated by the agent threads when they need to replace a dirty block in the buffer pool. We call these *eviction-synchronous writes*, or simply *synchronous writes*. In other cases, such as when pages are flushed by the page cleaners, the eviction of the block is not imminent, and in fact may not occur at all. We call these *eviction-asynchronous writes*, or simply *asynchronous writes*. Note that the distinction between synchronous and asynchronous writes and the distinction between replacement and recoverability writes are essentially orthogonal.

Assuming that the storage server could somehow make these distinctions, what kinds of hints could it take from write requests? We present several cases here.

- **synchronous writes:** A synchronous write of block b indicates that b is about to be evicted from the storage client's cache. If the storage server chooses to place b into its cache, it can be confident that b is not also in the storage client's cache.
- **asynchronous replacement writes:** An asynchronous replacement write of block b indicates two things. First, b is present in the storage client's cache. Second, the storage client is preparing b for eventual eviction, although eviction may not be imminent. Thus, in this case, it is not obvious what the storage server should infer from the occurrence of the write. However, we observe that if the storage client is well-designed, an asynchronous replacement write does suggest that b is quite likely to be evicted from the storage client cache in the near future. This is a weaker hint than that provided by a synchronous write. However, given that a well-designed client will seek to avoid synchronous writes, asynchronous replacement write hints may ultimately be more useful because they are more frequent.
- **asynchronous recoverability writes:** An asynchronous recoverability write of block b indicates that b is present in the storage client's cache and that it may have been present there for some time, since recoverability writes should target old unwritten updates. Unlike an asynchronous replacement write, a recoverability write of block b does not indicate that b 's eviction from the storage client cache is imminent, so b is a poor candidate for placement in the storage server cache.

To exploit these hints, it is necessary for the storage server to distinguish between these different types of writes. One possibility is for the server to attempt to infer the type of write based on the information carried in the write request: the source of the block, the destination of the block in the storage server, or the contents of the block. Another alternative is for the storage client to determine the type of each write and then label each write with its type for the benefit of the storage server. This is the approach that we have taken. Specifically, we propose that the storage client associate a *write hint* with each write request that it generates. A write hint is simply a tag with one of three possible values: SYNCH, REPLACE, or RECOV. These tags correspond to the three cases described earlier.

The necessity of tagging means that the use of write hints is not entirely transparent to the storage client. Thus, under the classification proposed by Chen et al [5], write hints would be considered to be an “aggressively collaborative” technique, although they would be among

the least aggressive techniques in that category. On the positive side, only a couple of bits per request are required for tagging, a negligible overhead. More importantly, we believe that it should be relatively easy and natural to identify write types from within the storage client. As noted in Section 5, we easily instrumented DB2 Universal Database to label each write with one of the three possible write types described above. Moreover, the types of write requests that we consider are not specific to DB2. Other major commercial database management systems, including Oracle [17] and Microsoft SQL Server [14], distinguish recoverability writes from replacement writes and try to do the writes asynchronously, resorting to synchronous writes only when necessary. Non-DBMS storage clients, such as file systems, also face similar issues. Finally, it is worth noting that the storage client does not need to understand how the storage server's cache operates in order to attach hints to its writes. Write hints provide information that may be useful to the storage server, but they do not specify how it should manage its cache.

4 Managing the Storage Server Cache

In this section, we discuss using the write hints introduced in Section 3 to improve the performance of second-tier cache replacement policies. We present techniques for extending two important cache replacement policies (LRU and MQ) so that they take advantage of write hints. We also present a new cache replacement algorithm that relies primarily on the information provided by write hints. But first, we address the question of how write hints can be used to achieve the goals of second-tier cache management.

4.1 Using Hints for Cache Management

Our goals in managing the second-tier cache are twofold. We want to maintain *exclusiveness* between the first- and second-tier caches, which means that the second tier should not cache blocks that are already cached in the first tier. At the same time, we want the second tier to cache blocks that will eventually be useful for the first tier. These are blocks whose re-reference distance (defined as the number of requests in the I/O stream between successive references to the block) is beyond the locality that could be captured in the first tier, and so will eventually miss in the first tier.

When choosing blocks to cache in the second tier, we should bear in mind that hits in the second tier are only useful for *read* requests from the first tier, but not write requests. Thus, the second-tier cache management policy should try to cache blocks that will cause read misses in the first tier.

We should also bear in mind that the second tier does not have to cache every block that is accessed by the first tier. The storage server could choose not to cache a block that is accessed, but rather to send the block from the storage device directly to the storage client (on a read miss), or from the client directly to the device (on a write).² This is different from other caching scenarios (e.g., virtual memory) in which the cache manager must cache every block that is accessed. Thus, storage server cache management has an extra degree of flexibility when compared to other kinds of cache management: when a new block arrives and the cache is full, the cache manager can evict a block to make room for the new block, or it can choose *not to cache the new block*.

With these points in mind, we consider the information provided by SYNCH, REPLACE, and RECOV write requests and also by read requests (which we label READ). SYNCH and REPLACE writes of a block b indicate that the block will be evicted from the first tier, so they provide hints that b should be cached in the second tier, with SYNCH providing a stronger hint than REPLACE. Caching b in the second tier will not violate exclusiveness, and future read accesses to b , which most likely will miss in the first tier, will hit in the second tier.

Conversely, a READ request for block b indicates that b will have just been loaded into the first-tier cache. We cannot determine from the READ request how long b will be retained in the first-tier cache. If recency-of-use plays a role in the storage client's cache management decisions, then we can expect that b will be a very poor candidate for caching at the storage server, as it is likely to remain in the client's cache for some time. On the other hand, the client's cache manager may take factors besides recency-of-use into account in deciding to evict b quickly. For example, if b is being read as part of a large sequential table scan performed by a database system then b may be quickly evicted from the client, and potentially re-referenced.

RECOV writes provides little information to the storage server cache. On the one hand, the written block is known to be in the storage client cache, which makes it a poor candidate for caching at the server. On the other hand, a RECOV write of b indicates that b has probably been in the storage client cache for a long time. Thus, the RECOV write does not provide as strong a negative hint as a READ.

Next, we illustrate how two important cache replacement policies (LRU and MQ) can be extended to take advantage of hints, and we present a new algorithm that relies primarily on request types (i.e., hints) to manage the cache.

4.2 LRU+Hints

We extend the least recently used (LRU) cache replacement policy by using hints to manage the LRU list and to decide whether or not to cache accessed blocks. We consider a simple extension: we cache blocks that occur in SYNCH or REPLACE write requests, since such blocks are likely to be evicted from the storage client cache. Blocks that occur in RECOV write requests or READ requests are not added to the cache.

Specifically, in the case of a SYNCH or REPLACE write for block b , we add b to the cache if it is not there and we move it to the most-recently-used (MRU) end of the LRU list. If a replacement is necessary, the LRU block is replaced. In the case of a RECOV or READ request for block b , we make no changes to the contents of the cache or to the recency of the blocks, except during cold start, when the cache is not full. During cold start, RECOV and READ blocks are cached and placed at the LRU end of the LRU list. Of course, in the case of a READ request, the server checks whether the requested block is in its cache, and it serves the requested block from the cache in case of a hit. This hint-aware policy is summarized in Algorithm 1.

4.3 MQ+Hints

The Multi-Queue (MQ) [21] algorithm is a recently proposed cache replacement algorithm designed specifically for second-tier cache management. It has been shown to perform better than prior cache replacement algorithms, including other recently proposed ones such as ARC [13] and LIRS [10]. The algorithm uses multiple LRU queues, with each queue representing a range of reference frequencies. Blocks are promoted to higher frequency queues as they get referenced more frequently, and when we need to evict a block, we evict from the lower frequency queues first. Thus, MQ chooses the block for eviction based on a combination of recency and frequency.

To implement its eviction policy, MQ tracks the recency and frequency of references to the blocks that are currently cached. MQ also uses an auxiliary data structure called the *out queue* to maintain statistics about some blocks that have been evicted from the cache. Each entry in the out queue records only the block statistics, not the block itself, so the entries are relatively small. The out queue has a maximum size, which is a configurable parameter of the MQ policy, and it is managed as an LRU list.

We extend the MQ algorithm with hints in the same way in which we extended LRU. If a request is a SYNCH or REPLACE, we treat it exactly as it would be treated under the original MQ algorithm. If the request is a READ,

Algorithm 1 LRU+Hints

LRUWITHHINTS(b : block access)

```
1  if  $b$  is already in the cache /* cache hit */
2    then if  $type(b) = \text{SYNCH}$  or  $type(b) = \text{REPLACE}$ 
3        then move  $b$  to the MRU end of the LRU list;
4  elseif  $type(b) = \text{SYNCH}$  or  $type(b) = \text{REPLACE}$  /* cache miss */
5    then insert  $b$  at the MRU end of the LRU list, evicting the LRU block to make room if needed;
6  elseif cache is not full /* cache miss and not SYNCH or REPLACE */
7    then insert  $b$  at the LRU end of the LRU list;
```

we check the queues for a hit as usual. However, the queues are not updated at all unless the cache is not full, in which case the block is added as it would be under the original algorithm. RECOV requests are ignored completely unless the cache is not full, in which case the block is added as in the original algorithm.

4.4 The TQ Algorithm

In this section, we present a new cache replacement algorithm that relies primarily on request types, as indicated by write hints, to make replacement decisions. We call this algorithm the *type queue (TQ) algorithm*. Among our hint-aware algorithms, TQ places the most emphasis on using request types (or hints) for replacement. We show in Section 5 that the TQ algorithm outperforms other candidate algorithms. TQ is summarized in Figure 3 and Algorithm 2.

As described earlier, blocks that occur in SYNCH and REPLACE write requests are good candidates for caching at the storage server, since there is a good chance that they will soon be evicted from the storage client. Blocks that are requested in READ requests are not likely to be requested soon, although we can not be certain of this. The TQ policy accounts for this by caching READ requests at the server, but at lower priority than SYNCH and REPLACE requests. Thus, if a block is read, we will retain it in the storage server cache if possible, but not at the expense of SYNCH or REPLACE blocks. RECOV writes provide neither a strong positive hint to cache the block (since the block is known to be at the client) nor a strong negative hint that the block should be removed from the server's cache. To reflect this, the TQ policy effectively ignores RECOV writes.

The TQ algorithm works by maintaining two queues for replacement. A high priority queue holds cached blocks for which the most recent non-RECOV request was a SYNCH or REPLACE write. A low priority queue holds cached blocks for which the most recent non-RECOV request was a READ. When a SYNCH or REPLACE request for block b occurs, b is added to the high

priority queue if b is not cached, or moved to the high priority queue if it is in the low priority queue. If b is in the high priority queue and a READ request for b occurs, then it is moved to the low priority queue. Thus, the sizes of these two queues are not fixed, and will vary over time depending on the request pattern. Replacements, when they are necessary, are always made from the low priority queue unless that queue is empty. If the low priority queue is empty, then replacements are made from the high priority queue.

RECOV writes are ignored, which means that they do not affect the contents of the cache or the order of the blocks in the two queues. The only exception to this is during cold start, when the cache is not full. During cold start, blocks that occur in RECOV write requests are added to the low priority queue if they are not already in the cache.

The low priority queue is managed using an LRU policy.³ The high priority queue, which is where we expect most read hits to occur, is managed using a replacement policy that we call *latest predicted read*, or LPR. When block b is placed into the high priority queue (because of a REPLACE or SYNCH write to b), the TQ algorithm makes a prediction, $nextReadPosition(b)$, of the time at which the next READ request for b will occur. When block replacement in the high priority queue is necessary, the algorithm replaces the block b with the latest (largest) $nextReadPosition(b)$.

This policy is similar in principle to the optimal off-line policy. However, unlike the off-line policy, LPR must rely on an imperfect prediction of $nextReadPosition(b)$. To allow it to make these predictions, the TQ algorithm maintains an estimate of the expected *write-to-read distance* of each block, which is the distance (number of cache requests) between a REPLACE or SYNCH write to the block and the first subsequent READ request for the block. When block b is added to the high priority queue, $nextReadPosition(b)$ is set to the current cache request count plus the expected write-to-read distance for b .

The TQ policy uses a running average of all the past

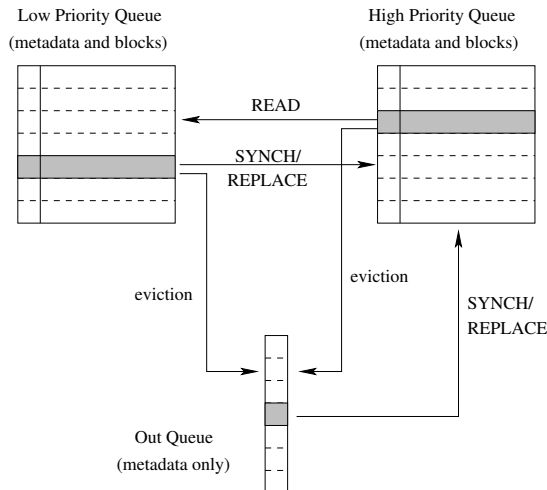


Figure 3: Structures used by TQ. Arrows show possible movements between queues in response to cache requests.

write-to-read distances of a block as its estimate of the expected write-to-read distance of this block. The policy maintains this running average of write-to-read distances for each block in the cache. In addition, like the MQ policy, TQ maintains an auxiliary data structure in which it tracks write-to-read distances and other reference statistics for a limited number of blocks that have previously been in the cache but have been evicted. For consistency with the terminology used by MQ, we call this data structure the TQ *out queue*. The maximum number of entries in the out queue is a parameter to the TQ algorithm. When an eviction from the out queue is necessary, the entry with the largest write-to-read distance is evicted.

When a block is added to the cache, TQ checks the out queue for an entry containing reference statistics about this block. If the block is found in the out queue, its write-to-read distance is obtained from the out queue, and the entry for the block is then removed from the out queue. If the block is not found in the out queue, its expected write-to-read distance in the cache is assumed to be infinite.

To maintain the running average of write-to-read distances for the blocks in the cache, TQ tracks the cache request count of the last REPLACE or SYNCH write request to each block in the high priority queue of the cache. This is done whether the request is bringing a new block into the cache, or whether it is a hit on a block already in the cache. When a READ request is a hit on a block in the high priority queue of the cache, the distance between this read and the most recent REPLACE or SYNCH request to this block is computed. The running average of write-to-read distances for this block is updated to in-

clude this new write-to-read distance.

When a block is evicted from the cache, an entry recording the expected write-to-read distance and the position of the most recent REPLACE or SYNCH write of this block is added to the out queue, and the out queue entry with the highest write-to-read distance is evicted to make room if necessary.

5 Evaluation

We used trace-driven simulations to evaluate the performance of the cache management techniques described in Section 4. The goal of our evaluation is to determine whether the use of write hints can improve the performance of the storage server cache. We also studied the performance of an optimal cache management technique to determine how much room remains for improvement.

5.1 Methodology

For the purposes of our evaluation, we used DB2 Universal Database (version 8.2) as the storage system client. We instrumented DB2 so that it would record traces of its I/O requests. We also modified DB2 so that it would record an appropriate write hint with each I/O request that it generates. These hints are recorded in the I/O trace records.

To collect our traces, we drove the instrumented DB2 with a TPC-C [19] OLTP workload, using a scale factor of 25. The initial size of the database, including all tables and indexes, is 606,317 4KB blocks, or approximately 2.3 Gbytes. The database grows slowly during the simulation run. The I/O request stream generated by DB2 depends on the settings of a variety of parameters. Table 1 shows the settings for the most significant parameters. We studied DB2 buffer pools ranging from 10% of the (initial) size of the database to 90% of the database size. The `softmax` and `chnpggs_thresh` parameters are important because they control the mix of write types in the request stream. The `chnpggs_thresh` gives the percentage of buffer pool pages that must be dirty to cause the page cleaners to begin generating replacement writes to clean them. The `softmax` parameter defines an upper bound on the amount of log data that will have to be read after a failure to recover the database. Larger values of `softmax` allow longer recovery times and result in fewer recoverability writes by the page cleaners. By fixing `chnpggs_thresh` at 50% (near DB2's default value) and varying `softmax`, we are able to control the mix of replacement and recoverability writes generated by the page cleaners.

Table 2 summarizes the traces that we collected and used for our evaluation. The 300.400 trace is our baseline trace, collected using our default DB2 parameter set-

Algorithm 2 The TQ Algorithm

TQACCESS(b : block access)

```
1  /* for the sake of simplicity, this assumes that the cache and the out queue are already full */
2  if type( $b$ ) = READ
3      then if  $b$  is in  $Q_{high}$  /*  $b$  is in high priority queue */
4          then move  $b$  to  $Q_{low}$ ; /* move  $b$  to low priority queue */
5          /* if this READ follows a SYNCH or REPLACE, update write-to-read distance */
6          if  $b$  is in cache or  $Q_{out}$  and  $lastWritePosition(b) > 0$ 
7              then update  $avgWriteReadDist(b)$  using ( $currentPosition - lastWritePosition(b)$ );
8                   $lastWritePosition(b) = 0$ ;
9  elseif type( $b$ ) = SYNCH or type( $b$ ) = REPLACE
10     then if  $b$  is in  $Q_{low}$ 
11         then  $nextReadPosition(b) = currentPosition + avgWriteReadDist(b)$ ;
12             move  $b$  to  $Q_{high}$ ; /* move  $b$  to high priority queue */
13              $lastWritePosition(b) = currentPosition$ ; /* remember when this write happened */
14     elseif  $b$  is in  $Q_{out}$  and  $lastWritePosition = 0$ 
15         then  $nextReadPosition(b) = currentPosition + avgWriteReadDist(b)$ ;
16             move  $victim$  from cache to  $Q_{out}$ ;
17             /* victim is LRU in  $Q_{low}$ , or latest  $nextReadPosition$  in  $Q_{high}$  if  $Q_{low}$  is empty */
18             move  $b$  to  $Q_{high}$ ; /* put  $b$  into high priority queue */
19              $lastWritePosition(b) = currentPosition$ ; /* remember when this write happened */
20     elseif  $b$  is not in  $Q_{high}$  and  $b$  is not in  $Q_{out}$ 
21         then remove  $Q_{out}$  entry with largest  $avgWriteReadDist$ ;
22             move  $victim$  from cache to  $Q_{out}$ ;
23             /* victim is LRU in  $Q_{low}$ , or latest  $nextReadPosition$  in  $Q_{high}$  if  $Q_{low}$  is empty */
24             put  $b$  into  $Q_{high}$ ; /* put  $b$  into high priority queue */
25              $lastWritePosition(b) = currentPosition$ ; /* remember when this write happened */
26              $nextReadPosition(b) = \infty$ ;
```

Parameter	Our Default Value	Other Values	Description
bufferpool size	300000 4KB blocks	60000, 540000 blocks	size of the DBMS buffer pool
softmax	400	50, 4000	recovery effort threshold
chnpggs_thresh	50%	-	buffer pool dirtiness threshold
maxagents	1000	-	maximum number of agent threads
num_iocleaners	50	-	number of page cleaner threads

Table 1: DB2 Parameter Settings

Trace Name	Buffer Pool Size in blocks	softmax	Number of Requests	Synch. Writes	Asynchronous Replacement Writes	Asynchronous Recoverability Writes	Reads
300_400	300K (1.1 GB)	400	13269706	0.00%	62.57%	3.60%	33.83%
60_400	60K (234 MB)	400	15792519	0.08%	48.89%	0.18%	50.85%
540_400	540K (2.1 GB)	400	12238848	0.00%	35.78%	49.89%	14.33%
300_4000	300K (1.1 GB)	4000	13226138	0.01%	65.37%	0.11%	34.51%
300_50	300K (1.1 GB)	50	15175377	0.00%	0.03%	74.33%	25.64%

Table 2: I/O Request Traces

tings. The remaining traces were collected using alternative buffer pool sizes and `softmax` settings. Not surprisingly, increasing the size of the DB2 buffer pool decreases the percentage of read requests in the trace (because more read requests hit in the buffer pool). Large buffer pools also tend to increase the frequency of recoverability writes, since updated pages tend to remain in the buffer pool longer. As discussed above, smaller values of `softmax` increase the prevalence of recoverability writes. The 300_50 trace represents a fairly extreme scenario with a very low `softmax` setting. This causes DB2 to issue a recoverability write soon after a page has been updated, so that recovery will be extremely fast. Although these settings are unlikely to be used in practice, we have included this trace for the sake of completeness.

We used these traces to drive simulations of a storage server buffer cache running the various algorithms described in Section 4. In addition, we implemented a variation of the off-line MIN algorithm [4], which we call OPT, as a means of establishing an upper bound on the hit ratio that we can expect in the storage server's buffer. Suppose that a storage server cache with capacity C has just received a request for block b . The OPT algorithm works as follows:

- If the cache is not full, put b into the cache.
- If the cache is full and it includes b , leave the cache contents unchanged.
- If the cache is full and it does not include b , then from among the C blocks currently in the cache plus b , eliminate the block that will not be *read* for the longest time. Keep the C remaining blocks in the cache.

Note that this algorithm may choose *not* to buffer b at all if it is advantageous to leave the contents of the cache unchanged.

For the MQ, MQ+Hints, and TQ algorithm, we set maximum number of entries in the out queue to be equal to the number of blocks that fit into the server's buffer cache. Thus, for each of these algorithms, the server tracks statistics for the pages that are currently buffered, plus an equal number of previously buffered pages. We subtracted the space required for the out queue from the available buffer space for each of these algorithms so that our comparisons with LRU and LRU+Hints, which do not require an out queue, would be on an equal-space basis.

On each simulation run, we first allow the storage server's cache to fill. Once the cache is warm, we then measure the *read hit ratio* for the storage server cache. This is the percentage of read requests that are found in the cache.

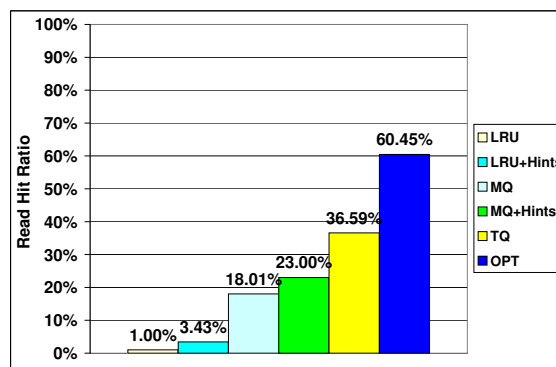


Figure 4: Read Hit Ratios in Storage Server Cache. Baseline (300_400) trace. Storage client cache size is 300K blocks (1.1 Gbytes), storage server cache size is 120K blocks (469 Mbytes).

5.2 Results: Baseline Case

Figure 4 shows the read hit ratios of the storage server cache under each of the techniques described in Section 4 for the baseline 300_400 trace with the storage server cache size set to 120K blocks (469 Mbytes). These results show that the LRU policy has very poor performance, which is consistent with other previous evaluations of LRU in second-tier caches [15, 21]. The LRU+Hints algorithm, which takes advantage of write hints, results in a hit ratio more than three times that of LRU, but it is still very low in absolute terms. The MQ algorithm, which considers frequency as well as recency, performs significantly better than LRU, and MQ+Hints improves on hint-oblivious MQ. The write hint based TQ algorithm provides the best performance, with a hit ratio nearly double that of MQ. TQ achieves more than half of the hit ratio of the off-line OPT algorithm.

5.3 Results: Sensitivity Analysis

We evaluated the sensitivity of the baseline results in Figure 4 to changes in three significant parameters: the size of the storage server cache, the size of the storage client cache (i.e., the DBMS buffer pool), and the value of the `softmax` parameter, which controls the mix of write types among the I/O requests.

Figure 5 shows the read hit ratio of the storage server cache as its size varies from 60K blocks (234 Mbytes) to 300K blocks (1.1 Gbytes), which is the size of the first-tier cache. Several observations can be made about these data. First, the relative advantage of the TQ algorithm is consistent until the server's cache reaches the largest size (300K blocks, 1.1 Gbytes) that we considered, at which point the advantage of TQ begins to diminish. For this large cache size, the improvement obtained by adding hints to MQ also becomes negligible.

However, for large cache sizes the performance of the simple LRU+Hints algorithm is much better than that of plain LRU, and comparable to that of TQ and the MQ policies. As the storage server cache gets smaller, the performance of LRU+Hints (and plain LRU) drops off quickly.

Figure 6 illustrates the impact of changing the storage client (DBMS) cache size, with the storage server cache size fixed at 120K blocks (469 Mbytes). These results show that management of the storage server cache becomes more difficult as the storage client cache becomes larger. Large storage client caches absorb most of the locality available in the request stream, leaving little for the storage server cache to exploit. Larger storage client caches also make it more difficult to maintain exclusiveness between the client and server caches. For very large client caches, the TQ algorithm performs more than five times better than the best hint-oblivious algorithm. However, *all* of the algorithms, including TQ, have poor performance in absolute terms, with read hit ratios far below that of the off-line OPT algorithm. When the storage client buffer is very small (60K blocks), all of the algorithms provide similar performance. In this case, the small storage client cache leaves temporal locality for the storage server cache to exploit, so that the difference between LRU and the remaining algorithms is not as great as it is when the client's cache is large.

Finally, Figure 7 shows the server cache read hit ratios as the `softmax` parameter increases from 50 to 4000. When `softmax` is very large (4000), the DBMS is effectively being told that long recovery times are acceptable. Under those conditions (trace 300_4000), the DBMS generates almost no recoverability writes; this is the primary difference between the baseline 300_400 trace and the 300_4000 trace. This has little impact on the performance of any of the algorithms.

At a `softmax` setting of 50, all of the hint-based algorithms, have similar performance, which is better than that of MQ and much better than LRU. When `softmax` is 50, almost three quarters of the I/O requests are recoverability writes, and there are no replacement writes. As was noted earlier, this represents an extreme scenario in which changes are flushed to the storage server almost immediately. As a result, this `softmax` setting generally gives poor overall system performance because of the substantial I/O write bandwidth that it requires, and is unlikely to be used in practice.

6 Related Work

Classical, general-purpose replacement algorithms, such as LRU and LFU, rely on the recency and frequency of requests to each block to determine which blocks to replace. More recent general-purpose algorithms, such

as 2Q [12], LRU-k [16], ARC [13], and CAR [3] improve on these classical algorithms, usually by balancing recency and frequency when making replacement decisions. Special purpose algorithms have been developed for use in database management systems [7] and other kinds of applications that cache data.

While any of the general-purpose algorithms can be used at any level of a cache hierarchy, researchers have recognized that cache management at the lower tiers of a hierarchy poses particular challenges, as was noted in Section 1. Zhou et al observed that access patterns at second tier caches are quite different from those at the first tier [21]. Muntz and Honeyman found that the second-tier cache in a distributed file system had low hit ratios because of this problem [15]. A second problem, pointed out by Wong and Wilkes, is that lower tier caches may contain many of the same blocks as upper tier caches [20]. This lack of exclusiveness wastes space and hurts the overall performance of the hierarchy.

Several general approaches to the problem of managing caches at the lower tiers in a hierarchy have been proposed. Since there is little temporal locality available in requests to second-tier caches, one strategy is to use a general-purpose replacement policy that is able to consider request frequency in addition to recency. Zhou, Philbin, and Li propose the multi-queue (MQ) algorithm (Section 4.3) to address this problem [22].

Although the MQ algorithm has been shown to be a better choice than LRU for managing a second-tier cache, the algorithm itself is not sensitive to the fact that it is operating in a hierarchy. Much of the work on caching in hierarchies focuses instead on techniques that are explicitly aware that they are operating in a hierarchy. One very simple technique of this type is to quickly remove from a lower-tier cache any block that is requested by an upper-tier, so that the block will not be cached redundantly [8, 5]. Other techniques involve tracking or simulating, at the second tier, certain aspects of the operation of the first-tier cache. One example of this is eviction-based caching, proposed by Chen, Zhou, and Li [6]. Under this technique, the second-tier cache tracks the target memory location of every block read by the first tier. This identifies where in the first tier cache each cached block has been placed. When the second-tier observes a new block being placed in the same location as a previously-requested block, it infers that the previously-requested block has been evicted from the first-tier cache and should be fetched into the second-tier cache. This places an extra load on the storage system, because it speculatively prefetches blocks.

The X-RAY mechanism takes a similar approach [2]. However, X-RAY assumes that the first tier is a file system, and it takes “gray-box” approach [1] to inferring the contents of the file system's cache. X-RAY can distin-

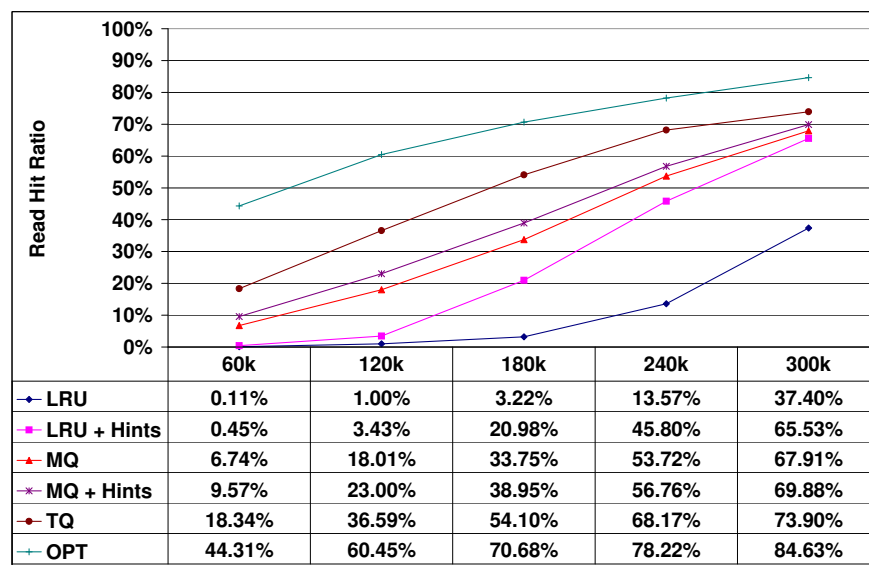


Figure 5: Read Hit Ratios in the Storage Server Cache. Baseline (300_400) trace. Storage client cache size is 300K blocks, storage server cache size varies from 60K blocks to 300K blocks.

guish file meta-data (i-nodes) from file data. It inspects the meta-data when it is flushed to the tier-two cache, and it uses the resulting information (e.g., access and update timestamps) to predict which blocks are likely to be in the file system’s cache. Sivathanu et al proposed a related technique called semantically-smart disks [18]. Like X-RAY, this assumes that the first tier is a file system. A probe process running against the file system allows the disk system at the second tier to discover, e.g., which blocks hold file system meta-data. It then uses this information to improve caching performance in the disk system.

All of the techniques discussed above share the property that they are transparent to the first-tier cache, i.e., they can be deployed without modifying the code that manages the first tier. Chen et al called these techniques “hierarchically aware” [5]. Other techniques, called “aggressively collaborative” by the same authors, require some modification to the first-tier. Wong and Wilkes defined a DEMOTE operation that is issued by the first tier cache to send evicted blocks to the second tier [20]. This operation can be used to achieve the same effect as eviction-based caching, except that with DEMOTE it is not necessary for the second-tier to infer the occurrence of first-tier evictions. Another possibility is for the first tier to pass hints to the second tier. For example, Chen et al describe Semantics-Directed Caching, in which the first-tier cache provides hints to the second tier about the importance (to the first tier) of blocks that it requests [5]. Franklin et al propose a technique for collaboratively managing the caches at a database client and a database server, in which the client passes a hint to the

server before it evicts a block, and the server can then ask the client to send it the block on eviction if the client has the only cached copy of this block [8].

The write hints proposed in this paper belong to the general class of “aggressively collaborative” techniques. However, they are complementary to previously proposed techniques of this class. For example, we could still exploit demotion information [20] or other kinds of hints [8, 5] while using write hints.

Another approach for managing two or more tiers of caches in a hierarchy is to use a single, unified controller. The Unified and Level-aware Caching (ULC) protocol controls a cache hierarchy from the first tier by issuing RETRIEVE and DEMOTE commands to caches at the lower tiers to cause them to move blocks up and down the cache hierarchy [11]. Zhou, Chen, and Li describe a similar approach, which they call “global” L2 buffer cache management, for a two-level hierarchy [5].

7 Conclusion

In this paper we observe that write hints can provide useful information that can be exploited by a storage server to improve the efficiency of its cache. We propose hint-aware versions of two existing hint-oblivious replacement policies, as well as TQ, a new hint-based policy. Trace-driven simulations show that the hint-aware policies perform better than the corresponding hint-oblivious policies. Furthermore, the new policy, TQ, had the best performance under almost all of the conditions that we studied.

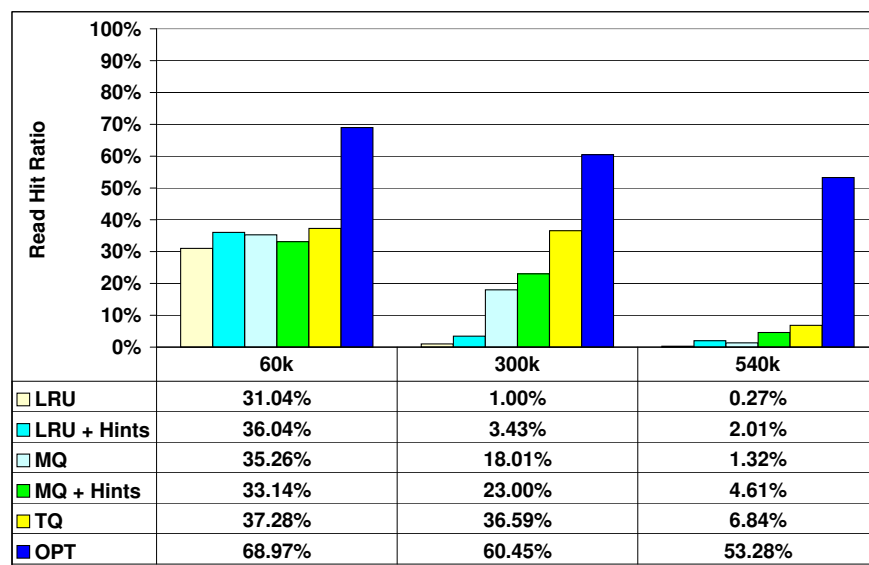


Figure 6: Read Hit Ratios in the Storage Server Cache. Traces 60_400, 300_400, and 540_400. Storage client cache size varies from 60K blocks (234 Mbytes) to 540K blocks (2.1 Gbytes). Storage server cache size is 120K blocks (469 Mbytes).

Our work focused on a configuration in which a DBMS, running an OLTP workload, acts as the storage client. In this common scenario, write hints are quite valuable to the storage server. The write hints themselves, however, are general, and reflect issues that must be faced by any type of storage client that caches data. Thus, we are optimistic that the benefits of write hints will extend to other types of storage clients that experience write-intensive workloads.

Possibilities for future work include investigating the use of write hints for other types of workloads or storage clients. They also include adding an aging mechanism to the TQ policy, and investigating avenues for the real world adoption of write hints, possibly through enhancements to the the SCSI interface.

8 Acknowledgments

We would like to thank Matt Huras and Calisto Zuzarte, from the IBM Toronto Lab, for their comments and assistance. This work was supported by IBM through the Center for Advanced Studies (CAS), and by Communications and Information Technology Ontario (CITO).

References

- [1] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, October 2001.
- [2] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-
- Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, June 2004.
- [3] S. Bansal and D. Modha. CAR: Clock with adaptive replacement. In *Proc. of the 3rd USENIX Symposium on File and Storage Technologies (FAST'04)*, March 2004.
- [4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [5] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 145–156, 2005.
- [6] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based cache placement for storage caches. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 269–282, June 2003.
- [7] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB)*, pages 127–141, August 1985.
- [8] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server database architectures. In *Proc. International Conference on Very Large Data Bases (VLDB'92)*, pages 596–609, 1992.
- [9] International Business Machines Corporation. *IBM DB2 Universal Database Administration Guide: Performance*, version 8.2 edition.
- [10] Song Jiang and Xiaodong Zhang. Lirs: An efficient low interference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'02)*, pages 31–42, June 2002.
- [11] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 168–177, March 2004.

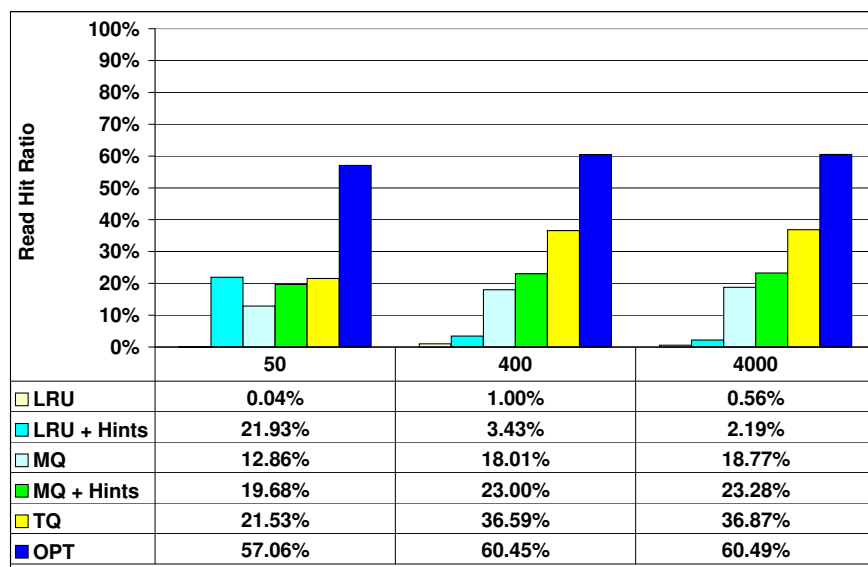


Figure 7: Read Hit Ratios in the Storage Server Cache as `softmax` is varied. Traces 300_50, 300_400, and 300_4000. Storage client cache size is 300K blocks (1.1 Gbytes), storage server cache size is 120K blocks (469 Mbytes).

- [12] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proc. International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, 1994.
- [13] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, March 2003.
- [14] Microsoft Corporation. *Microsoft SQL Server 2000 Books Online*.
- [15] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.
- [16] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, pages 297–306, 1993.
- [17] Oracle Corporation. *Oracle Database Concepts*, version 10g release 2 (10.2) edition, June 2005.
- [18] Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, March 2003.
- [19] Transaction Processing Performance Council. *TPC Benchmark C*, revision 5.4 edition, 2005.
- [20] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference (USENIX 2002)*, pages 161–175, June 2002.
- [21] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(7), July 2004.
- [22] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 91–104, June 2001.

Notes

¹We focus on two-tier cache hierarchies for clarity of presentation, but our discussion and proposed techniques extend to cache hierarchies with more than two tiers.

²Some short-term buffering may be required to accommodate transfer speed mismatches and request bursts. We have ignored this for the sake of simplicity.

³We expect that hits in the low priority queue will be uncommon, and that the behavior of the TQ policy will not be very sensitive to the replacement policy in the low priority queue.

WOW: Wise Ordering for Writes – Combining Spatial and Temporal Locality in Non-Volatile Caches

Binny S. Gill and Dharmendra S. Modha

IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120

Emails: {binnyg,dmodha}@us.ibm.com

Abstract—Write caches using fast, non-volatile storage are now widely used in modern storage controllers since they enable hiding latency on writes. Effective algorithms for write cache management are extremely important since (i) in RAID-5, due to read-modify-write and parity updates, each write may cause up to four separate disk seeks while a read miss causes only a single disk seek; and (ii) typically, write cache size is much smaller than the read cache size – a proportion of 1 : 16 is typical.

A write caching policy must decide: what data to destage. On one hand, to exploit *temporal locality*, we would like to destage data that is least likely to be re-written soon with the goal of minimizing the total number of destages. This is normally achieved using a caching algorithm such as LRW (least recently written). However, a read cache has a very small uniform cost of replacing any data in the cache, whereas the cost of destaging depends on the state of the disk heads. Hence, on the other hand, to exploit *spatial locality*, we would like to destage writes so as to minimize the average cost of each destage. This can be achieved by using a disk scheduling algorithm such as CSCAN, that destages data in the ascending order of the logical addresses, at the higher level of the write cache in a storage controller. Observe that LRW and CSCAN focus, respectively, on exploiting either temporal or spatial locality, but not both simultaneously. We propose a new algorithm, namely, Wise Ordering for Writes (WOW), for write cache management that effectively combines and balances temporal and spatial locality.

Our experimental set-up consisted of an IBM xSeries 345 dual processor server running Linux that is driving a (software) RAID-5 or RAID-10 array using a workload akin to Storage Performance Council's widely adopted SPC-1 benchmark. In a cache-sensitive configuration on RAID-5, WOW delivers peak throughput that is 129% higher than CSCAN and 9% higher than LRW. In a cache-insensitive configuration on RAID-5, WOW and CSCAN deliver peak throughput that is 50% higher than LRW. For a random write workload with nearly 100% misses, on RAID-10, with a cache size of 64K, 4KB pages (256MB), WOW and CSCAN deliver peak throughput that is 200% higher than LRW. In summary, WOW has better or comparable peak throughput to the best of CSCAN and LRW across a wide gamut of write cache sizes and workload configurations. In addition, even at lower throughputs, WOW has lower average response times than CSCAN and LRW.

I. INTRODUCTION

Over the last three decades, processor speeds have increased at an astounding average annual rate of 60%. In contrast, disks which are electro-mechanical devices have improved their access times at a comparatively

meager annual rate of about 8%. Moreover, disk capacity grows 100 times per decade, implying fewer available spindles for the same amount of storage [1]. These trends dictate that a processor must wait for increasingly larger number of cycles for a disk read/write to complete. A huge amount of performance literature has focused on hiding this I/O latency for disk bound applications.

Caching is a fundamental technique in hiding I/O latency and is widely used in storage controllers (IBM Shark, EMC Symmetrix, Hitachi Lightning), databases (IBM DB2, Oracle, SQL Server), file systems (NTFS, EXT3, NFS, CIFS), and operating systems (UNIX variants and Windows). SNIA (www.snia.org) defines a cache as “A high speed memory or storage device used to reduce the effective time required to read data from or write data to a lower speed memory or device.” We shall study cache algorithms in the context of a storage controller wherein fast, but relatively expensive, random access memory is used as a cache for slow, but relatively inexpensive, disks. A modern storage controller's cache typically contains volatile memory used as a *read cache* and a non-volatile memory used as a *write cache*.

Read cache management is a well studied discipline, for a survey and for some recent work, see [2], [3], [4], [5]. There are a large number of cache replacement algorithms in this context, see, for example, LRU, CLOCK, FBR, LRU-2, 2Q, LRFU, LIRS, MQ, ARC, and CAR. In contrast, write caching is a relatively less developed subject. Here, we shall focus on algorithms for write cache management in the context of a storage controller equipped with fast, non-volatile storage (NVS).

A. Fast Writes using NVS: When to Destage

For early papers on the case for and design of write cache using NVS, see [6], [7], [8], [9], [10]. Historically, NVS was introduced to enable fast writes.

In the absence of NVS, every write must be synchronously written (destaged) to disk to ensure consistency, correctness, durability, and persistence. Non-volatility enables *fast writes* wherein writes are stored safely in the cache, and destaged later in an asynchronous fashion thus hiding the write latency of the disk. To guarantee continued low latency for writes, the

data in the NVS must be drained so as to ensure that there is always some empty space for incoming writes; otherwise, follow-on writes will become effectively synchronous, impacting adversely the response time for writes. On the other hand, if the writes are drained very aggressively, then one cannot fully exploit the benefits of write caching since average amount of NVS cache utilized will be low. Reference [11] introduced a linear threshold scheduling scheme that varies the rate of destages based on the instantaneous occupancy of the write cache. Other simpler schemes include least-cost scheduling and scheduling using high/low mark [11], [12], [13]. Another related issue is the size of a write burst that the write cache is designed to absorb while providing the low response time of fast writes. The write destage policy needs to ensure that a corresponding portion of NVS is available on an average. In Section IV-E, we describe a novel and effective approach to tackle this problem by using adaptive high/low watermarks combined with linear threshold scheduling.

B. A Fundamental Decision: What to Destage

In this paper, we shall focus on the central question of the order in which the writes are destaged from the NVS. Due to its asynchronous nature, the contents of the write cache may be destaged in any desired order without being concerned about starving any write requests. As long as NVS is drained at a sufficiently fast rate, the precise order in which contents of NVS are destaged does not affect fast write performance. However, the decision of what to destage can crucially affect (i) the peak write throughput and (ii) the performance of concurrent reads.

The capacity of disks to support sequential or nearly sequential write traffic is significantly higher than their capacity to support random writes, and, hence, destaging writes while exploiting this physical fact can significantly improve the peak write throughput of the system. This was one of the fundamental motivations for development of log-structured file systems [14], [15] in a slightly different context. Thus, any good write caching algorithm should leverage sequentiality or spatial locality to improve the write throughput and hence the aggregate throughput of the system.

In the presence of concurrent reads, both the writes being destaged and the reads compete for the attention of the disk head. From the perspective of the applications, writes represent a background load on the disks and, indirectly, make read response times higher and read throughput lower. Less obtrusive the writes, the lesser the response time for the reads. We argue that the effect of writes on reads is significant:

- The widely accepted storage benchmark SPC-1 [16], [17] contains 60% writes and 40% reads.

While fast writes to NVS enable low response times for writes, these writes, when destaged, do interfere with the reads, increasing the read response times.

- When RAID is used at the back-end, writes are significantly more expensive than the reads. For example, for RAID-5, a read miss on a page may cause only one disk head to move, whereas due to read-modify-write and due to parity updates a write destage may cause up to four separate disk seeks. Similarly, for RAID-10, a write destage causes two disk heads to move.
- NVS is commonly built using battery-backed volatile RAM. The size of NVS is limited by the life of the battery which should be sufficient to dump the contents to a non-battery-backed non-volatile store like disks in case of a power failure. A write cache size of one-sixteenth of the read cache size is not unusual. Given the small relative size of the NVS, it tends to produce relatively fewer write hits, and, hence, a significantly large fraction of writes must be destaged. Further, unlike reads, writes do not benefit significantly from client side caching.

The first two arguments imply that for the SPC-1 benchmark using RAID-5 ranks, writes constitute at least six times the load of the read misses on the back-end! This underscores the tremendous influence that writes have on read performance and the peak throughput of the system. In summary, improving the order in which writes are destaged can significantly improve the overall throughput and response times of the storage controller.

C. Our Contributions

In read caches, the cost of evicting any page from the cache is the same and very small. Hence, the objective of a read cache is simple: to minimize the miss ratio. In contrast, we propose that the performance of a write destage algorithm depends upon two factors: (i) the total number of destages to disks, namely, the write miss ratio and (ii) the average cost of each destage. Roughly speaking, the objective of write destage algorithms is to minimize the product of the above two terms. Minimizing this product would result in the highest peak write throughput in absence of any reads. Even in presence of concurrent reads, this product attempts to minimize the amount of time that the disk heads are occupied in serving writes leading to minimizing the average read response time, while maximizing aggregate throughput.

To minimize the first term, the algorithm should exploit *temporal locality*, namely, should destage data that is least likely to be written to amongst all data in the write cache. To minimize the second term, the algorithm

should exploit *spatial locality* and should destage data that are closer on the disks together so as to exploit the position of the heads and the geometry of the disks in the system for higher throughput.

Classically, in read caches, LRU (least recently used) policy has been used to exploit temporal locality. The analogous policy for exploiting temporal locality in writes is known as LRW that destages the least-recently written page [18], [19], [20].

Algorithms for exploiting spatial locality have been studied in the context of disk scheduling. Many such algorithms require a detailed knowledge of the instantaneous position of the disk head, and exploit the precise knowledge of the location of each data relative to the disk head. In this paper, we are working in the context of a storage controller from which most of the disk parameters are hidden by the underlying RAID and disks. Hence, generally, speaking, spatial locality is hard to exploit at such upper memory levels, see, for example, [21]. We argue, however, that one of the algorithms, namely, CSCAN, that destages data in the ascending order of the logical addresses, can be reasonably successfully applied at even upper levels of memory hierarchy. We empirically demonstrate that as the size of NVS managed by CSCAN increases, the throughput of the system increases for a wide variety of workloads.

The destage order suggested by LRW and CSCAN are generally different, hence, it is only possible to exploit temporal locality or spatial locality, but not both. To emphasize, one reduces number of disk seeks which is the goal of caching, while the other reduces the cost of each disk seek which is the goal of scheduling. This brings us to the main focus of this paper: a novel algorithm that combines both temporal and spatial locality. As our main contribution, we combine an approximation to LRU, namely, CLOCK, with CSCAN to construct a new, simple-to-implement algorithm, namely, Wise Ordering for Writes (WOW), that effectively achieves this goal. The key new idea is to maintain a recency bit akin to CLOCK in CSCAN, and to skip destages of data that has been recently written to.

To demonstrate effectiveness of WOW, we used the following hardware, storage, and workload. The hardware was an IBM xSeries 345 dual processor server running Linux equipped with 4GB RAM that was used as NVS. The storage consisted of 5 disks. Using software RAID, we created a RAID-5 array in 4 data disks + 1 parity disk configuration. We also created a RAID-10 array in 2 + 2 configuration. As the workload, we employed an earlier implementation of the Storage Performance Council's SPC-1 benchmark that is now extremely widely used by many vendors of storage

systems [16], [17]. We refer to our workload as SPC-1 Like. In a set-up with a high degree of temporal locality ("a cache-sensitive configuration"), WOW delivers peak throughput that is 129% higher than CSCAN and 9% higher than LRW. In a set-up with very little temporal locality ("a cache-insensitive configuration"), WOW and CSCAN deliver peak throughput that is 50% higher than LRW. For a random write workload with nearly 100% misses, on RAID-10, with a cache size of 64K, 4KB pages, WOW and CSCAN deliver peak throughput that is 200% higher than LRW. Similarly, for the random write workload with nearly 100% misses, on RAID-5, with a cache size of 16K, 4KB pages, WOW and CSCAN deliver peak throughput that is 147% higher than LRW. In summary, WOW has better or comparable peak throughput to the best of CSCAN and LRW across a wide gamut of write cache sizes and workload configurations. In addition, even at lower throughputs, WOW has lower average response times than CSCAN and LRW. In another experiment, using SPC-1 Like workload, as cache size is varied, we explore both cache-insensitive and cache-sensitive regimes. We clearly show that CSCAN is good for cache-insensitive regimes, LRW is good for cache-sensitive regimes, whereas WOW is evergreen and is good across the whole range of cache sizes. In summary, WOW is a practical algorithm that fundamentally enhances the capacity of a storage controller to perform more I/Os.

D. Outline of the Paper

In Section II, we briefly survey previous related research, and we argue the utility of CSCAN for exploiting spatial locality even at upper levels of cache hierarchy. In Section III, we present the new algorithm WOW. In Section IV, we describe the experimental set-up. In Section V, we describe the workloads. In Section VI, we present our main quantitative results. Finally, in Section VII, we conclude with the main findings of this paper.

II. PRIOR WORK

A. Temporal Locality

Algorithms for exploiting temporal locality have been studied extensively in the context of read caches. Several state-of-the-art algorithms include LRU, CLOCK, FBR, LRU-2, 2Q, LRFU, LIRS, MQ, ARC, and CAR. For a detailed review of these algorithms, please see some recent papers [2], [4], [5].

These algorithms attempt to reduce the miss ratio. However, as explained in Section I-C, in write caching, it is not sufficient to minimize the miss ratio alone, but we must also pay attention to the average cost of destages. The latter factor is completely ignored by the above

algorithms. We will demonstrate in the paper that a write caching algorithm has a higher hit ratio than some other algorithm and yet the second algorithm delivers a higher throughput. In other words, decreasing the miss ratio without taking into account its effect on the spatial locality is unlikely to guarantee increased performance. Thus, the problem of designing good write caching algorithms is different from that of designing algorithms for read caching.

In this paper, we focus on **LRW** as the prototypical temporal locality algorithm. We also exploit a simple approximation to **LRU**, namely, **CLOCK** [22], that is widely used in operating systems and databases. **CLOCK** is a classical algorithm, and is a very good approximation to **LRU**. For a recent paper comparing **CLOCK** and **LRU**, see [4].

B. Spatial Locality

The key observation is that given the geometry and design of the modern day disks, sequential bandwidth of the disks is significantly higher (for example, more than 10 times) than its performance on 100% random write workload. The goal is to exploit this differential by introducing spatial locality in the destage order.

This goal has been extensively studied in the context of disk scheduling algorithms. The key constraint in disk scheduling is to ensure fairness or avoid starvation that occurs when a disk I/O is not serviced for an unacceptably long time. In other words, the goal is to minimize the average response time and its variance. For detailed reviews on disk scheduling, see [23], [24], [25], [26]. A number of algorithms are known: First-come-first-serve (**FCFS**) [27], Shortest seek time first (**SSTF**) [28] serves the I/O that leads to shortest seek, Shortest access time first (**SATF**), **SCAN** [28] serves I/Os first in increasing order and then in decreasing order of their logical addresses, Cyclical **SCAN** (**CSCAN**) [29] serves I/Os only in the increasing order of their logical addresses. There are many other variants known as **LOOK** [30], **VSCAN** [31], **FSCAN** [27], Shortest Positioning Time First (**SPTF**) [26], **GSTF** and **WSTF** [24], and Largest Segment per Track **LST** [11], [32].

In the context of the present paper, we are interested in write caching at an upper level in the memory hierarchy, namely, for a storage controller. This context differs from disk scheduling in two ways. First, the goal is to maximize throughput without worrying about fairness or starvation for writes. This follows since as far as the writer is concerned, the write requests have already been completed after they are written in the **NVS**. Furthermore, there are no reads that are being scheduled by the algorithm. Second, in disk scheduling, detailed knowledge of the disk head and the exact position of various outstanding writes relative to this head position

and disk motion is available. Such knowledge cannot be assumed in our context. For example, [21] found that applying **SATF** at a higher level was not possible. They concluded that "... we found that modern disks have too many internal control mechanisms that are too complicated to properly account for in the disk service time model. This exercise lead us to conclude that software-based **SATF** disk schedulers are less and less feasible as the disk technology evolves." Reference [21] further noted that "Even when a reasonably accurate software-based **SATF** disk scheduler can be successfully built, the performance gain over a **SCAN**-based disk scheduler that it can realistically achieve appears to be insignificant ...". The conclusions of [21] were in the context of single disks, however, if applied to **RAID-5**, their conclusions will hold with even more force.

For these reasons, in this paper, we focus on **CSCAN** as the fundamental spatial locality algorithm that is suitable in our context. The reason that **CSCAN** works reasonably well is that at anytime it issues a few outstanding writes that all fall in a thin annulus on the disk, see, Figure 1. Hence, **CSCAN** helps reduce seek distances. The algorithm does not attempt to optimize for further rotational latency, but rather trusts the scheduling algorithm inside the disk to order the writes and to exploit this degree of freedom. In other words, **CSCAN** does not attempt to outsmart or outguess the disk's scheduling algorithm, but rather complements it.

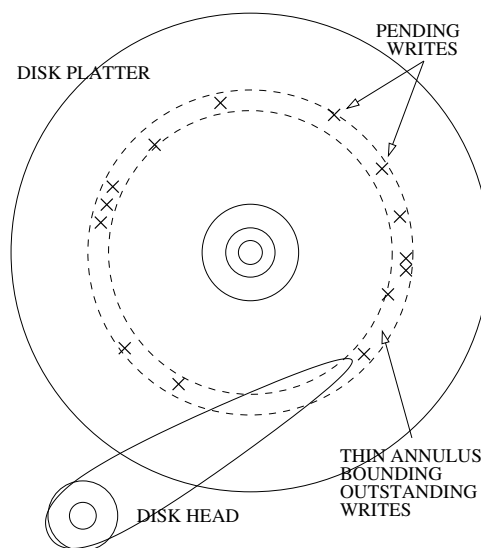


Fig. 1. A visual depiction of how **CSCAN** localizes the outstanding writes on a thin annulus on the disk platter.

In Figure 2, we demonstrate that as the size of **NVS** managed by **CSCAN** grows, so does the achieved throughput. We use a workload that writes 4KB pages randomly over a single disk, and that has nearly 100%

Random Write Workload
Single Disk with 8.875 million 4KB pages

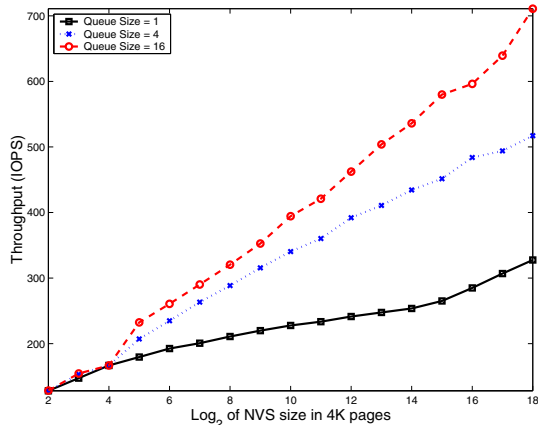


Fig. 2. A plot of the size of NVS used by CSCAN versus the throughput (in IOPS) achieved by the algorithm on a workload that writes 4KB pages randomly over a single disk. It can be seen that throughput seems to grow logarithmically as NVS size grows. Also, as the queue size, namely, the number of outstanding requests issued by CSCAN to the disk increases, the throughput also increases.

write misses. We also use several queue sizes that control the maximum number of outstanding writes that CSCAN can issue to the disk. It appears that throughput grows proportionally to the logarithm of the NVS size. As the size of NVS grows, assuming no starvation at the underlying disk, the average thickness of the annulus in Figure 1 should shrink – thus permitting a more effective utilization of the disk. Observe that the disks continuously rotate at a constant number of revolutions per second. CSCAN attempts to increase the number of writes that can be carried out per revolution.

C. Combining Temporal and Spatial Locality

The only previous work on the subject is [33], [32] that partitions the cache into a “hot” zone that is managed via LRW and a “cold” zone that is managed via LST. The authors point out several drawbacks of their approach: (i) “The work in this dissertation only deals with the interaction between the cache and one disk.” [32, p. 126] and (ii) “One of the most immediate aspects of this work requiring more research is the method to determine the size of the hot zone for the stack model-based replacement algorithm. We determined the best size for the hot zone empirically in our experiments.” [32, p. 125]. To continue this program further would entail developing an adaptive algorithm for tuning the size of the hot zone. However, note that the hot zone optimizes for temporal locality (say “apples”) and the cold zone for spatial locality (say “oranges”). It is not currently known how to compare and trade apples versus oranges to determine the best adaptive partition.

In this paper, we will present a natural combination of LRW and CSCAN that obviates this need, and yet delivers convincing performance.

III. WOW

A. Preliminaries

We are working in the context of a storage controller. Typically, a storage controller connects to a RAID controller which, in turn, connects to physical disks. We assume that there is no write cache at lower levels such as RAID and disks. In other words, there is no fast write at lower levels and I/O scheduling is limited to concurrent requests issued by the storage controller. Also, note that a typical RAID controller may choose to implement FCFS or such simpler scheduling algorithm, while an individual disk may implement a smarter disk scheduling algorithm such as SATF. We can make no assumptions about these algorithms at the level of a storage controller. Also, typically the amount of write cache in storage controllers per RAID array or per disk is much larger than the amount of cache in the RAID array or disks.

We will use 4KB pages as the smallest unit of cache management. We will divide each disk into strips, where each strip is a logically and physically contiguous set of pages. Here, we use 64KB as the strip size. In RAID, we will define a stripe as a collection of strips where each participating disk contributes one strip to a stripe. Due to mirroring in RAID-10 and parity in RAID-5, the effective storage provided by a stripe is less than its physical size.

The notion of a hit is straightforward in read caching, but is somewhat involved for write caching. In write caching, a hit can be a hit on a page, a hit on a strip, or a hit on a stripe. These different hits may have different payoffs, for example, in RAID-5, a page hit saves four seeks, whereas a stripe hit and a page miss saves two seeks because of shared parity. In RAID-5, we will manage cache in terms of stripe groups. In RAID-10, we will manage cache in terms of strip groups. This allows a better exploitation of temporal locality by saving seeks and also spatial locality by coalescing writes together. We will refer to a strip or stripe group as a *write group*.

B. WOW : The Algorithm

We now describe our main contribution which is a new algorithm that combines the strengths of CLOCK, a predominantly read cache algorithm, and CSCAN, an efficient write cache algorithm, to produce a very powerful and widely applicable write cache algorithm. See Figures 3 and 4 for a depiction of the data structures and the algorithm.

The basic idea is to proceed as in CSCAN by maintaining a sorted list of write groups. The smallest

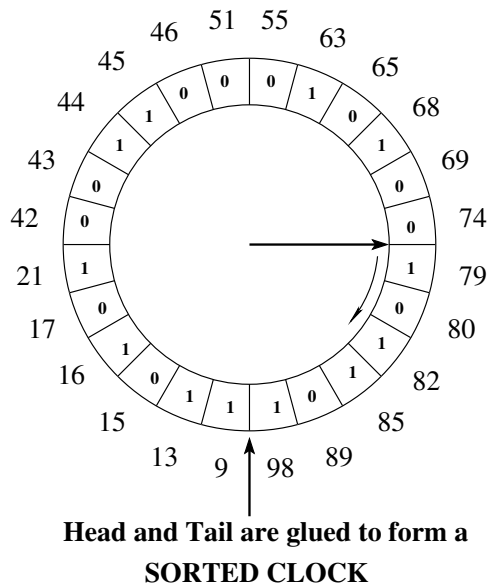


Fig. 3. A visual structure of the WOW algorithm. If the CLOCK's recency bit is ignored, the algorithm becomes CSCAN. The clock hand represents the *destagePointer* in Figure 4.

and the highest write groups are joined forming a circular queue. The additional new idea is to maintain a “recency” bit akin to CLOCK with each write group. The algorithm now proceeds as follows.

A write group is always inserted in its correct sorted position. Upon insertion, its recency bit is set to zero. However, on a write hit, the recency bit is set to one. The destage operation proceeds as in CSCAN, wherein a destage pointer is maintained that traverses the circular list looking for destage victims. In CSCAN every write group that is encountered is destaged. However, we only allow destage of write groups whose recency bit is zero. The write groups with a recency bit of one are skipped, however, their recency bit is turned off, and reset to zero. The basic idea is to give an extra life to those write groups that have been hit since the last time the destage pointer visited them. This allows us to incorporate recency representing temporal locality on one hand, and small average distance between consecutive destages representing spatial locality. The simplicity of the algorithm is intentional so that it succeeds in real systems. The superiority of the algorithm (demonstrated in Section VI) to the current state-of-the-art should encourage its widespread use.

C. WOW and its Parents

Since WOW is a hybrid between LRW or CLOCK, and CSCAN, we now contrast and compare these algorithms.

WOW is akin to CSCAN, since it destages in essentially the same order as CSCAN. However, WOW is

CACHE MANAGEMENT POLICY:

Page x in write group s is written:

```

1:  if ( $s$  is in NVS) // a write group hit
2:    if (the access is not sequential)
3:      set the recencyBit of  $s$  to 1
4:    endif
5:  if ( $x$  is in NVS) // a page hit
6:    set the recencyBit of  $s$  to 1
7:  else
8:    allocate  $x$  from FreePageQueue
      and insert  $x$  in  $s$ 
9:  endif
10: else
11:  allocate  $s$  from
      FreeStripeGroupHeaderQueue
12:  allocate  $x$  from FreePageQueue
13:  insert  $x$  into  $s$  and  $s$  into the sorted queue
14:  initialize the recencyBit of  $s$  to 0
15:  if ( $s$  is the only write group in NVS)
16:    initialize the destagePointer to point to  $s$ 
17:  endif
18: endif

```

DESTAGE POLICY:

```

19: while (needToDestage())
20:   while (the recencyBit of the write group
      pointed to by the destagePointer is 1)
21:     reset the recencyBit to 0
22:     AdvanceDestagePointer()
23:   endwhile
24:   destage all pages in the write group pointed
      to by the destagePointer and
      move them to FreePageQueue
25:   move the destaged write group to
      FreeStripeGroupHeaderQueue
26:   AdvanceDestagePointer()
27: endwhile

28: AdvanceDestagePointer()
29:   if (destagePointer is pointing to the
      highest address write group in the queue)
30:     reset the destagePointer to point to the
      lowest address write group in the queue
31:   else
32:     advance the destagePointer to the next
      higher address write group in the queue
33:   endif

```

Fig. 4. The WOW Algorithm.

different from **CSCAN** in that it skips destage of data that have been recently written to in the hope that that are likely to be written to again. **WOW** generally will have a higher hit ratio than **CSCAN** at the cost of an increased gap between consecutive destages.

WOW is akin to **LRW** in that it defers write groups that have been recently written. Similarly, **WOW** is akin to **CLOCK** in that upon a write hit to a write group a new life is granted to it until the destage pointer returns to it again. **WOW** is different from **CLOCK** in that the new write groups are not inserted immediately behind the destage pointer as **CLOCK** would but rather in their sorted location. Thus, initially, **CLOCK** would always grant one full life to each newly inserted write group, whereas **WOW** grants on an average half that much time. **WOW** generally will have a significantly smaller gap between consecutive destages than **LRW** at the cost of a generally lower hit ratio.

D. Is RAID just one Big Disk?

Intuitively, the reader may wish to equate the destage pointer in **WOW** to a disk head. It is as if **WOW** or **CSCAN** are simulating the position of the disk head, and destaging accordingly. In practice, this intuition is not strictly correct since (i) concurrent read misses may be happening which can take the disk heads to arbitrary locations on disks; and (ii) the position of the heads cannot be strictly controlled, for example, due to read-modify-write in RAID-5; and (iii) at a lower level, either the RAID controller or the individual disks may re-order concurrent write requests. In view of these limitations, the purpose of **WOW** or **CSCAN** is to spatially localize the disk heads to a relatively narrow region on the disks with the idea that the resulting disk seeks will be less expensive than random disk seeks which may move the head across a larger number of cylinders on the disks. In practice, we have seen that these observations indeed hold true.

E. WOW Enhancements

We anticipate that **WOW** will engender a class of algorithms which modify **WOW** along multiple dimensions. We have shown how to combine **LRW** and **CSCAN**. Another important feature of workloads that indicates temporal locality is “frequency”. It is possible to incorporate frequency information into **WOW** by utilizing a counter instead of just a recency bit. It is extremely interesting and challenging to pursue adaptive variants of **WOW** that dynamically adapt the balance between temporal and spatial locality. Furthermore, it will be interesting to see if a marriage of **MQ**, **ARC**, **CAR**, etc. algorithms can be consummated with **CSCAN** to develop algorithms that separate out recency from frequency to further enhance the power of **WOW**.

Another aspect of temporal locality is the duration for which a new stripe of page is allowed to remain in the cache without producing a hit. For simplicity, we have chosen the initial value of the recency bit to be set to 0 (see line 14 in Figure 4). Thus, on an average, a new write group gets a life equal to half the time required by the destage pointer to go around the clock once. If during this time, it produces a hit, it is granted one more life until the destage pointer returns to it once again. If the initial value is set to 1, then—on an average—a new write group gets a life equal to 1.5 times the time required by the destage pointer to go around the clock once. More temporal locality can be discovered if the initial life is longer. However, this happens at the cost of larger average seek distances as more pages are skipped by the destage head. It may be possible to obtain the same effect without the penalty by maintaining a history of destaged pages in the spirit of **MQ**, **ARC**, and **CAR** algorithms.

F. WOW : Some design points

1) *Sequential Writes*: It is very easy to detect whether a write group is being written to by a sequential access write client or a random access one, see, for example, [3, Section II.A]. In this paper, we enhance the algorithm in Figure 4 to never set the recency bit to 1 on a sequential access. This is reflected in lines 2-4 in Figure 4. This heuristic gives the bulky sequential stripes a smaller life and frees up the cache for more number of less populated stripes that could potentially yield more hits.

2) *Variable I/O sizes*: In Figure 4, we have dealt with an illustrative wherein a single page x in a write group s is written to. Typical write requests may write to a variable number of pages that may lie on multiple write groups. Our implementation correctly handles these cases via simple modifications to the given algorithm.

3) *Data Structures*: The algorithm requires very simple data structures: a sorted queue for storing write groups, a hash-based lookup for checking whether a write group is presented in the sorted queue (that is for hit/miss determination), and a destage pointer for determining the next candidate write group for destage. The fact that insertion in a sorted queue is an $O(\log(n))$ operation does not present a practical problem due to the limited sizes of NVS and the availability of cheap computational power.

IV. EXPERIMENTAL SET-UP

We now describe a system that we built to measure and compare the performance of the different write cache algorithms under a variety of realistic configurations.

A. The Basic Hardware Set-up

We use an IBM xSeries 345 machine equipped with two Intel Xeon 2 GHz processors, 4 GB DDR, and six 10K RPM SCSI disks (IBM, 06P5759, U160) of 36.4 GB each. A Linux kernel (version 2.6.11) runs on this machine hosting all our applications and standard workload generators. We employ five SCSI disks for the purposes of our experiments, and the remaining one for the operating system, our software, and workloads.

B. Storage: Direct Disks or Software RAID

We will study three basic types of configurations corresponding to I/Os going to either a single disk, to a RAID-5 array, or a RAID-10 array. In the first case, we issue direct (raw) I/O to one of the five disk devices (for example, `/dev/sdb`). In the latter two cases, we issue direct I/O to the virtual RAID disk device (for example, `/dev/md0`) created by using the Software RAID feature in Linux.

Software RAID in Linux implements the functionality of a RAID controller within the host to which the disks are attached. As we do not modify or depend on the particular implementation of the RAID controller, a hardware RAID controller or an external RAID array would work equally well for our experiments. We created a RAID-5 array using 5 SCSI disks in 4 data disk + 1 parity disk configurations. We chose the strip size (chunk size) for each disk to be 64KB. Resulting stripe group size was 256KB. Similarly, we created a RAID-10 array using 4 SCSI disks in 2 + 2 configuration. Once again, we chose the strip size for each disk to be 64KB.

C. NVS and Read Cache

We employ the volatile DDR memory as our write cache or NVS. The fact that this memory is not battery-backed does not impact the correctness or relevance of our results to real-life storage controllers. We shall, therefore, still refer to this memory as NVS. The write cache is implemented in shared memory and is managed by user space libraries that are linked to all the applications that refer to this shared memory cache. The size of the NVS can be set to any size up to the maximum size of the shared memory. This approach provides tremendous flexibility for our experiments by allowing us to benchmark various algorithms across a large range of NVS sizes.

For our experiments, we do not use a read cache as all disk I/Os are direct (or raw) and bypass the Linux buffer caches. This helps us eliminate an unnecessary degree of freedom for our experiments. Recall that read misses must be served concurrently, and disrupt the sequential destaging operation of WOW and CSCAN. Also, read misses compete for head time, and affect even LRW.

Eliminating the read cache serves to maximize read misses, and, hence, our setup is the most adversarial for NVS destage algorithms. In real-life storage controllers equipped with a read cache, the aggregate performance will depend even more critically on the write caching algorithm and thus magnify even further the performance differences between these algorithms.

A side benefit of maintaining a write cache is the read hits that it produces. The write caching algorithms are not intended to improve the read hit ratio primarily because the read cache is larger and more effective in producing read hits. Nevertheless, in our setup we do check the write cache for these not-so-numerous read hits and return data from the write cache on a hit for consistency purposes.

D. The Overall Software System

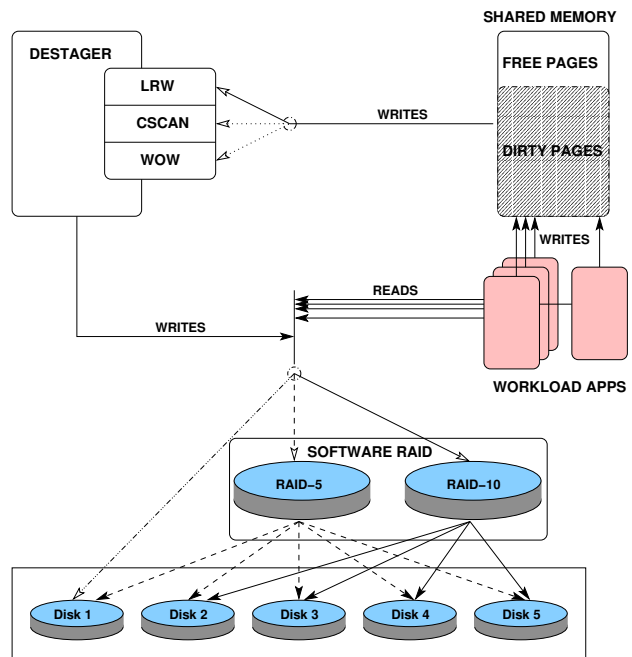


Fig. 5. The overall design of our experimental software system. The system is layered: (i) applications; (ii) NVS (shared memory) whose size can be varied from “cache-insensitive” (very small) to “cache-sensitive” (relatively large); (iii) destager which may choose to implement LRW, CSCAN, or WOW; and (iv) backend storage which may be a single disk, a RAID-5 array, or a RAID-10 array.

A schematic description of the experimental system is given in Figure 5.

The workload applications may have multiple threads that can asynchronously issue multiple simultaneous read and write requests. We implemented the write cache in user space for the convenience of debugging. As a result, we do have to modify the workload applications to replace the “`read()`” and “`write()`” system calls with our versions of these after linking our shared

memory management library. This is a very simple change that we have implemented for all the workload applications that we used in this paper.

For a single disk, NVS was managed in units of 4KB pages. For RAID-5, NVS was managed in terms of 256KB stripe write groups. Finally, in RAID-10, NVS was managed in terms of 64KB strip write groups. Arbitrary subsets of pages in a write group may be present in NVS. Write requests that access consecutively numbered write groups are termed *sequential*.

On a read request, if all requested pages are found in NVS, it is deemed a read hit, and is served immediately (synchronously). If, however, all requested pages are not found in NVS, it is deemed a read miss, and a synchronous stage (fetch) request is issued to the disk. The request must now wait until the disk completes the read. Immediately upon completion, the read request is served. As explained above, the read misses are not cached.

On a write request, if all written pages are found in NVS, it is deemed a write hit, and the write request is served immediately. If some of the written pages are not in NVS but enough free pages are available, once again, the write request is served immediately. If, however, some of the written pages are not in NVS and enough free pages are not available, the write request must wait until enough pages become available. In the first two cases, the write response time is negligible, whereas in the last case, the write response time can become significant. Thus, NVS must be drained so as to avoid this situation if possible.

We have implemented a user-space destager program that chooses dirty pages from the shared memory and destages them to the disks. This program is triggered into action when the free pages in the shared memory are running low. It can be configured to destage either to a single physical disk or to the virtual RAID disk. To decide which pages to destage from the shared memory, this program can choose between the three cache management algorithms: LRW, CSCAN, and WOW.

E. Queue Depth and When To Destage

To utilize the full throughput potential of a RAID array or even a single disk, it is crucial to issue multiple concurrent writes. This gives more choice to the scheduling algorithm inside the disks which, by design, usually tries to maximize the throughput without starving any I/Os. Furthermore, in RAID, the number of outstanding concurrent writes roughly dictates the number of disks heads that can be employed in parallel. The number of outstanding concurrent writes constitute a queue. As this queue length increases, both the throughput and the average response time increases. As

the queue length increases, the reads suffer, in that, they may have to wait more on an average. We choose a value, MAXQUEUE (say 20), as the maximum of number of concurrent write requests to the disks, where a write request is a set of contiguous pages within one write group.

We now turn our attention to the important decision of “When to Destage” that is needed in line 19 of Figure 4. At any time, we dynamically vary the number of outstanding destages in accordance with how full the NVS actually is. We maintain a *lowThreshold* which is initially set to 80% of the NVS size, and a *highThreshold* which is initially set to 90% of the NVS size. If the NVS occupancy is below the *lowThreshold* and we were not destaging sequential write group, we stop all destages. However, if NVS occupancy is below the *lowThreshold* but the previous destage was marked sequential and the next candidate destage is also marked sequential, then we continue the destaging at a slow and steady rate of 4 outstanding destages at any time. This ensures that sequences are not broken and their spatial locality is exploited completely. Further, this also takes advantage of disks’ sequential bandwidth. If NVS occupancy is at or above the *highThreshold*, then we always go full throttle, that is, destage at the maximum drain rate of MAXQUEUE outstanding write requests. We linearly vary the rate of destage from *lowThreshold* to *highThreshold* in a fashion similar to [11]. The more full within this range the NVS gets, the faster the drain rate; in other words, the larger the number of outstanding concurrent writes. Observe that the algorithm will not always use the maximum queue depth. Writing at full throttle regardless of the rate of new writes is generally bad for performance. What is desired is simply to keep up with the incoming write load without filling up NVS. Convexity of throughput versus response time curve indicates that a steady rate of destage is more effective than a lot of destages at one time and very few at another. Dynamically ramping up the number of outstanding concurrent writes to reflect how full NVS is helps to achieve this steady rate. Always using full throttle destage rate leads to abrupt “start” and “stop” situation, respectively, when the destage threshold is exceeded or reached.

We add one more new idea, namely, we dynamically adapt the *highThreshold*. Recall that write response times are negligible as long as NVS is empty enough to accommodate incoming requests, and can become quite large if NVS ever becomes full. We adapt the *highThreshold* to attempt to avoid this undesirable state while maximizing NVS occupancy. We implement a simple adaptive back-off and advance scheme. The *lowThreshold* is always set to be *highThreshold* minus 10% of NVS size. We define *desiredOccupancyLevel*

to be 90% of the NVS size. The `highThreshold` is never allowed to be higher than `desiredOccupancyLevel` or lower than 10% of NVS size. We maintain a variable called `maxOccupancyObserved` that keeps the maximum occupancy of the cache since the last time it was reset. Now, if and when the NVS occupancy drops below the current `highThreshold`, we decrement the `highThreshold` by any positive difference between `maxOccupancyObserved` and `desiredOccupancyLevel` and we reset `maxOccupancyObserved` to the current occupancy level. We keep a note of the amount of destages that happen between two consecutive resettings of `maxOccupancyObserved` in the variable `resetInterval`. Of course, decrementing `highThreshold` hurts the average occupancy levels in NVS, and reduces spatial as well as temporal locality for writes. Thus, to counteract this decrementing force, if after a sufficient number of destages (say equal to `resetInterval`) the `maxOccupancyObserved` is lower than the `desiredOccupancyLevel`, then we increment `highThreshold` by the difference between `desiredOccupancyLevel` and `maxOccupancyObserved`, and we reset `maxOccupancyObserved` to the current occupancy level.

V. WORKLOADS

A. Footprint

While modern storage controllers can make available an immense amount of space, in a real-life scenario, workloads actively use only a fraction of the total available storage space known as the *footprint*. Generally speaking, for a given cache size, the larger the footprint, the smaller the hit ratio, and vice versa. We will use backend storage in two configurations: (i) *Full Backend* in which the entire available backend storage will be used and (ii) *Partial Backend* in which we will use 7.1 million 512 byte sectors. In RAID-5, we shall have effectively the storage capacity of four disks at the back-end, where Full Backend amount to 284 million 512 byte sectors. Similarly, for RAID-10, we shall have effectively the storage capacity of two disks at the back-end, where Full Backend amount to 142 million 512 byte sectors.

B. SPC-1 Benchmark

SPC-1 is a synthetic, but sophisticated and fairly realistic, performance measurement workload for storage subsystems used in business critical applications. The benchmark simulates real world environments as seen by on-line, non-volatile storage in a typical server class computer system. SPC-1 measures the performance of a storage subsystem by presenting to it a set of I/O operations that are typical for business critical applications like OLTP systems, database systems and mail server

applications. For extensive details on SPC-1, please see: [16], [17], [3]. A number of vendors have submitted SPC-1 benchmark results for their storage controllers, for example, IBM, HP, Dell, SUN, LSI Logic, Fujitsu, StorageTek, 3PARdata, and DataCore. This underscores the enormous practical and commercial importance of the benchmark. We used an earlier prototype implementation of SPC-1 benchmark that we refer to as SPC-1 Like.

SPC-1 has 40% read requests and 60% write requests. Also, with 40% chance a request is a sequential read/write and with 60% chance a request is a random read/write with some temporal locality. SPC-1 is a multi-threaded application that can issue multiple simultaneous read and writes. For a given cache/NVS size, the number of read/write hits produced by SPC-1 changes as the footprint of the backend storage changes. For example, for a given cache/NVS size, SPC-1 will produce more hits with a Partial Backend then with a Full Backend. Furthermore, it is easy to vary the target throughput in I/Os Per Second (IOPS) for the workload. Thus, it provides a very complete and versatile tool to understand the behavior of all the three write destage algorithms in a wide range of settings.

SPC-1's backend consists of three disjoint application storage units (ASU). ASU-1 represents a "Data Store", ASU-2 represents a "User Store", and ASU-3 represents a "Log/Sequential Write". Of the total amount of available back-end storage, 45% is assigned to ASU-1, 45% is assigned to ASU-2, and remaining 10% is assigned to ASU-3 as per SPC-1 specifications. In all configurations, we laid out ASU-3 at the outer rim of the disks followed by ASU-1 and ASU-2.

C. Random Write Workload

We will use a random write workload that uniformly writes 4KB pages over the Full Backend that is available. As long as the size of the cache is relatively smaller than the Full Backend size, the workload has little temporal locality, and will produce nearly 100% write misses. The think time, namely, the pause between completing one write request and issuing another one, of this workload is set to zero. In other words, this workload is capable, in principle, of driving a storage system at an infinite throughput. The throughput is limited only by the capacity of the system to serve the writes. This workload is extremely helpful in profiling behavior of the algorithms across a wide range of NVS sizes.

VI. RESULTS

A. LRW Does Not Exploit Spatial Locality

In Figure 6, we compare LRW, CSCAN, and WOW using random write workload (Section V-C) directed to

Random Write Workload (nearly 100% miss), Queue Depth = 20, Full Backend, RAID-10 (left panel), RAID-5 (right panel)

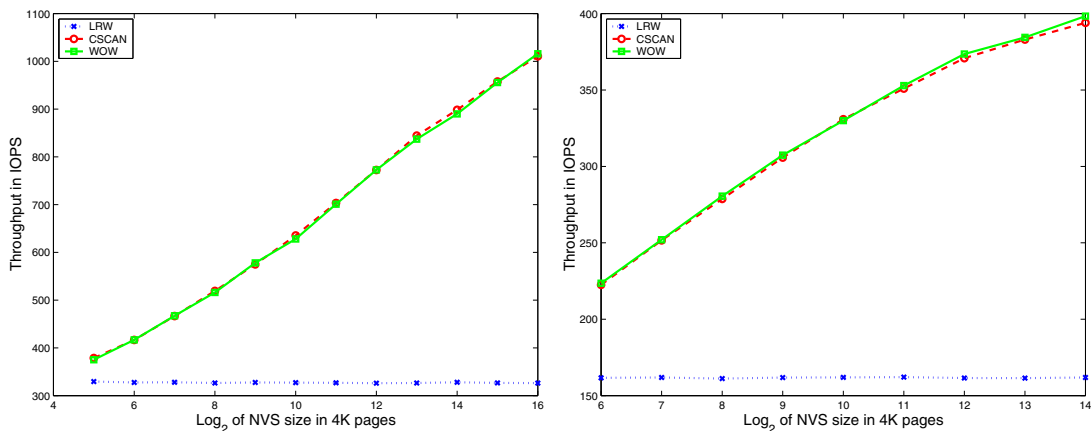


Fig. 6. A comparison of LRW, CSCAN, and WOW using random write workload using the Full Backend for both RAID-10 and RAID-5. It can be seen that the throughput of LRW does not depend upon the NVS size, whereas throughput of WOW and CSCAN exhibit a logarithmic gain as a function of the size of NVS.

Full Backend on RAID-5 and RAID-10. Since the workload has almost no temporal locality, the throughput of LRW remains constant as the NVS size increases. In contrast, WOW and CSCAN exhibit logarithmic gain in throughput as a function of the size of NVS by exploiting spatial locality (also see the related discussion in Section II-B). For RAID-10, at the lowest NVS size of 32 pages, WOW and CSCAN outperform LRW by 16%, while, quite dramatically, at the NVS size of 65,536 pages, WOW and CSCAN outperform LRW by 200%. Similarly, for RAID-5, at the lowest NVS size of 64 pages, WOW and CSCAN outperform LRW by 38%, while, quite dramatically, at the NVS size of 16,384 pages, WOW and CSCAN outperform LRW by 147%.

While, for brevity, we have shown results for a queue depth of 20. When we used a larger queue depth, performance of all three algorithms increased uniformly, producing virtually identical curves. Increasing queue depth beyond 128 in either RAID-10 or RAID-5 does not seem to help throughput significantly.

B. WOW is Good for Cache-sensitive and -insensitive Regimes

In Figure 7, we compare LRW, CSCAN, and WOW using SPC-1 Like workload (Section V-B) directed to Partial Backend on RAID-5 and RAID-10. We vary the size of NVS from very small (corresponding to cache-insensitive regime) to relatively large (corresponding to cache-sensitive regime). For RAID-10, a target throughput of 6000 IOPS was used for all NVS sizes. For RAID-5, a target throughput of 3000 IOPS was used for all NVS size except the largest one (100,000 pages) for which a target throughput of 6000 IOPS was used to drive the disks to full capacity.

In both graphs, it is easy to see that CSCAN dominates LRW in the cache-insensitive regime, but loses to LRW in the cache-sensitive regime. WOW, however, is evergreen, and performs well in both the regimes. This is easy to see since in a cache-insensitive regime CSCAN wins by exploiting spatial locality, whereas in cache-sensitive regime LRW puts up a nice performance by exploiting temporal locality. However, WOW which is designed to exploit both temporal and spatial locality performs well in both the regimes and across the entire range of NVS sizes.

Specifically, in RAID-10, for the smallest NVS size, in a favorable situation for CSCAN, WOW performs the same as CSCAN, but outperforms LRW by 4.6%. On the other hand, for the largest NVS size, in a favorable situation for LRW, WOW outperforms CSCAN by 38.9% and LRW by 29.4%.

Similarly, in RAID-5, for the smallest NVS size, in a favorable situation for CSCAN, WOW loses to CSCAN by 2.7%, but outperforms LRW by 9.7%. On the other hand, for the largest NVS size, in a favorable situation for LRW, WOW outperforms CSCAN by 129% and LRW by 53%. In Figure 8, we examine, in detail, the write hit ratios and average difference in logical address between consecutive destages for all three algorithms. The larger the write hit ratio the larger is the temporal locality. The larger the average distance between consecutive destages the smaller is the spatial locality. It can be seen that temporal locality is the highest for LRW, followed by WOW, and the least for CSCAN. On the contrary, spatial locality is the highest for CSCAN, followed by WOW, and the least (by 3 to 4 orders of magnitude) for LRW. As shown in the right panel of Figure 7, WOW outperforms both LRW and

SPC-1 Like Workload, Queue Depth = 20, Partial Backend, RAID-10 (left panel), RAID-5 (right panel)

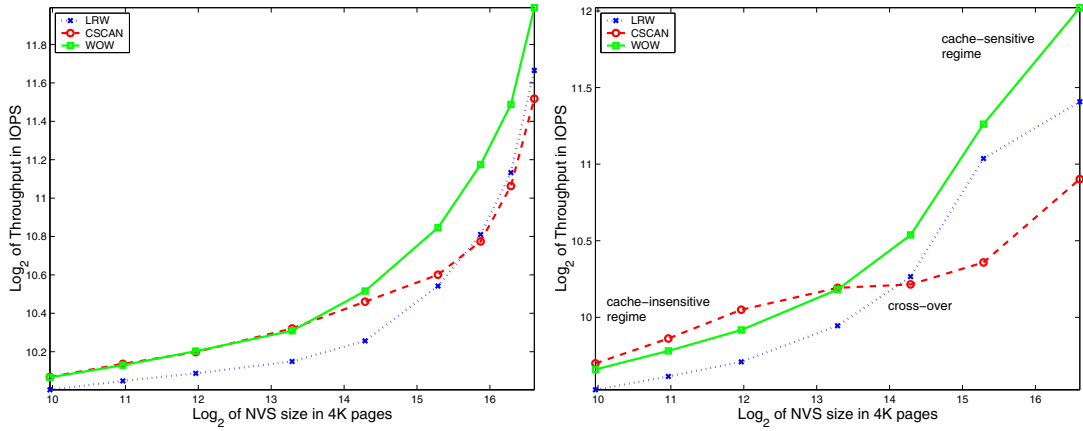


Fig. 7. A comparison of LRW, CSCAN, and WOW using SPC-1 Like workload using the Partial Backend for both RAID-10 and RAID-5. It can be seen that CSCAN is a good bet in cache-insensitive regime whereas LRW is an attractive play in cache-sensitive regime. However, WOW is attractive in both regimes.

SPC-1 Like Workload, RAID-5 with Partial Backend
Write Hit Ratio (left panel), Average Distance Between Consecutive Destages (right panel)

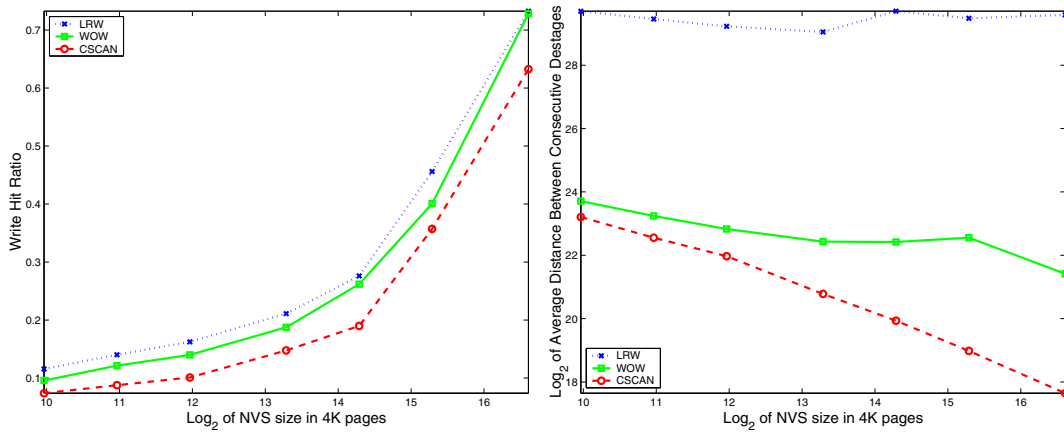


Fig. 8. The left panel shows the write hit ratios (indicating temporal locality) for LRW, CSCAN, and WOW, while using the SPC1-Like workload using partial backend for RAID-5. Temporal locality is the highest for LRW, followed by WOW, and the least for CSCAN. The right panel shows the corresponding average difference in logical address between consecutive destages for the three algorithms during the same run. The larger this average distance the smaller is the spatial locality. We can see that, contrary to the temporal locality, spatial locality is the highest for CSCAN, followed by WOW, and the least (by 3 to 4 orders of magnitude) for LRW. As shown in the right panel of Figure 7, WOW outperforms both LRW and CSCAN by effectively combining temporal as well as spatial locality.

CSCAN by effectively combining temporal as well as spatial locality.

C. Throughput versus Response Time in Cache-insensitive Scenario

In Figure 9, we compare LRW, CSCAN, and WOW using SPC-1 Like workload directed to Full Backend on RAID-5. We use an NVS size of 4K pages each of 4KB. Hence, NVS to backing store ratio is very low, namely, 0.011%, constituting a cache-insensitive scenario.

We vary the target throughput of SPC-1 Like from 100 IOPS to 1100 IOPS. At each target throughput, we

allow a settling time of 10 mins, after which we record average response time over a period of 5 minutes.

It can be clearly seen that WOW and CSCAN have virtually identical performances, and both significantly outperform LRW. In particular, it can be seen that LRW finds it impossible to support throughput beyond 515 IOPS. Demanding a target throughput higher than this point does not yield any further improvements, but rather worsens the response times dramatically. In contrast, WOW and CSCAN saturate, respectively, at 774 and 765 IOPS. In other words, WOW delivers a peak throughput that is 50% higher than LRW, and

SPC-1 Like, Cache-insensitive configuration, NVS size=4K pages, RAID-5, Full Backend

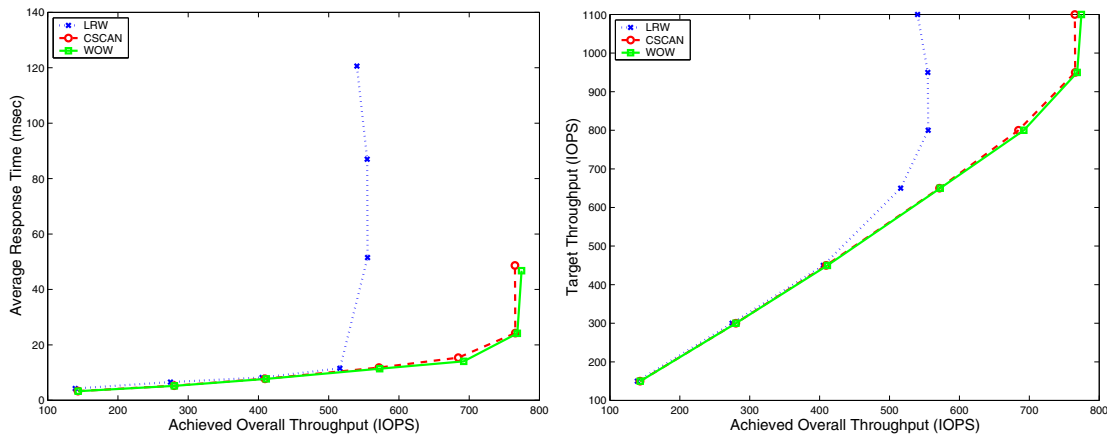


Fig. 9. A comparison of LRW, CSCAN, and WOW. The left panel displays achieved overall throughput versus achieved average response time. This set-up does not have much temporal locality. WOW and CSCAN have comparable performance, and both outperform LRW dramatically. Specifically, WOW increases the peak throughput over LRW by 50%. The right panel shows the target throughput corresponding to the data points in the left panel. It can be clearly seen that LRW hits an insurmountable stiff wall at a much lower throughput.

virtually identical to CSCAN.

D. Throughput versus Response Time in Cache-insensitive Scenario

In Figure 10, we compare LRW, CSCAN, and WOW using SPC-1 Like workload directed to Partial Backend on RAID-5. We use an NVS size of 40K pages each of 4KB. Hence, NVS to backing store ratio is relatively large, namely, 4.52%, constituting a cache-sensitive scenario.

We vary the target throughput of SPC-1 Like from 300 IOPS to 3000 IOPS. At each target throughput, we allow a settling time of 10 mins, after which we record average response time over a period of 8 minutes.

It can be clearly seen that WOW dramatically outperforms CSCAN and even outperforms LRW. In particular, it can be seen that CSCAN finds it impossible to support throughput beyond 1070 IOPS. In contrast, WOW and LRW saturate, respectively, at 2453 and 2244 IOPS. In other words, WOW delivers a peak throughput that is 129% higher than CSCAN, and 9% higher than LRW.

Remark VI.1 (backwards bending) Observe that in Figures 9 and 10 when trying to increase the target throughput beyond what the algorithms can support, the throughput actually drops due to increased lock and resource contention. This “backwards bending” phenomenon is well known in traffic control and congestion where excess traffic lowers throughput and increases average response time.

VII. CONCLUSIONS

It is known that applying sophisticated disk scheduling algorithms such as SATF at upper levels in cache hierarchy is a fruitless enterprise. However, we have demonstrated that CSCAN can be profitably applied even at upper levels of memory hierarchy for effectively improving throughput. As the size of NVS grows, for a random write workload, the throughput delivered by CSCAN seems to grow logarithmically for single disks, RAID-10, and RAID-5.

CSCAN exploits spatial locality and is extremely effective in cache-insensitive storage configurations. However, it does not perform as well in cache-sensitive storage configurations, where it loses to LRW that exploits temporal locality. Since, one cannot *a priori* dictate/assume either a cache-sensitive or a cache-insensitive scenario, there is a strong need for an algorithm that works well in both regimes. We have proposed WOW which effectively combines CLOCK (an approximation to LRW) and CSCAN to exploit both temporal locality and spatial locality.

We have demonstrated that WOW convincingly outperforms CSCAN and LRW in various realistic scenarios using a widely accepted benchmark workload. WOW is extremely simple-to-implement, and is ideally suited for storage controllers and for most operating systems. WOW fundamentally increases the capacity of a storage system to perform more writes while minimizing the impact on any concurrent reads.

REFERENCES

- [1] J. Gray and P. J. Shenoy, “Rules of thumb in data engineering,” in *ICDE*, pp. 3–12, 2000.

SPC-1 Like, Cache-sensitive configuration, NVS size=40K pages, RAID-5, Partial Backend

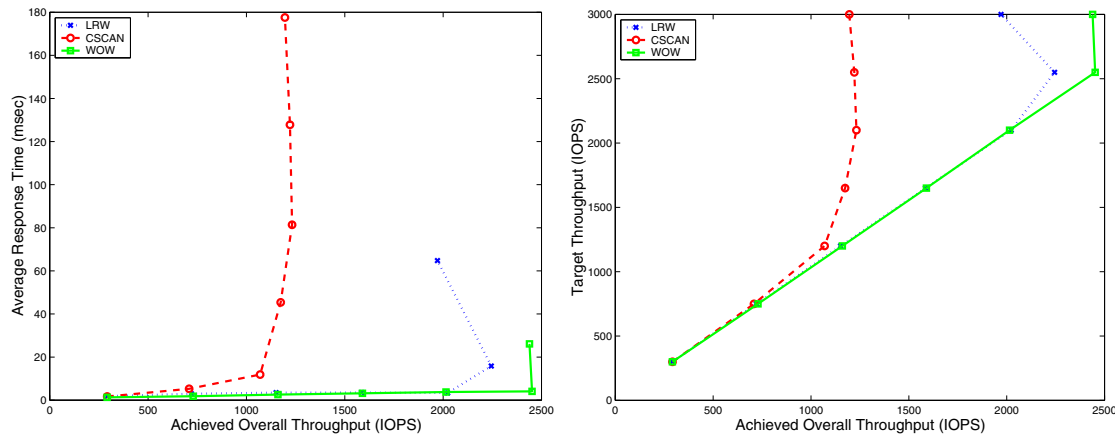


Fig. 10. A comparison of LRW, CSCAN, and WOW. The left panel displays achieved overall throughput versus achieved average response time. This set-up has significant temporal locality since the ratio of NVS size to the size of the backend is relatively high (4.52%). WOW increases the peak throughput over LRW by 9% and over CSCAN by 129%. The right panel shows the target throughput corresponding to the data points in the left panel. It can be clearly seen that CSCAN hits an insurmountable stiff wall at a much lower throughput.

- [2] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," *IEEE Computer*, vol. 37, no. 4, pp. 58–65, 2004.
- [3] B. S. Gill and D. S. Modha, "SARC: Sequential prefetching in adaptive replacement cache," in *USENIX*, 2005.
- [4] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *FAST 04*, pp. 142–163, 2004.
- [5] Y. Zhou, Z. Chen, and K. Li, "Second-level buffer cache management," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 505–519, 2004.
- [6] J. Menon and M. Hartung, "The IBM 3990 disk cache," in *Proc. IEEE Comput. Soc. Int. COMPCON Conf.*, 1988.
- [7] G. P. Copeland, T. Keller, R. Krishnamurthy, and M. Smith, "The case for safe RAM," in *Vldb*, pp. 327–335, 1989.
- [8] J. Menon, "Performance of RAID5 disk arrays with read and write caching," *Distributed and Parallel Databases*, vol. 2, no. 3, pp. 261–293, 1994.
- [9] J. Menon and J. Cortney, "The architecture of a fault-tolerant cached RAID controller," in *ISCA*, pp. 76–86, 1993.
- [10] K. Treiber and J. Menon, "Simulation study of cached RAID5 designs," in *HPCA*, pp. 186–197, 1995.
- [11] A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with nonvolatile caches," *IEEE Trans. Computers*, vol. 47, no. 2, pp. 228–235, 1998.
- [12] P. Biswas, K. K. Ramakrishnan, and D. Towsley, "Trace driven analysis of write caching policies for disk," *Performance Evaluation Review*, vol. 21, no. 1, pp. 12–23, Jun 1993.
- [13] Y. J. Nam and C. Park, "An adaptive high-low water mark destage algorithm for cached RAID5," in *PRDC*, pp. 177–184, 2002.
- [14] J. K. Ousterhout and F. Douglass, "Beating the I/O bottleneck: A case for log-structured file systems," *Operating Systems Review*, vol. 23, no. 1, pp. 11–28, 1989.
- [15] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.
- [16] B. McNutt and S. Johnson, "A standard test of I/O cache," in *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.
- [17] S. A. Johnson, B. McNutt, and R. Reich, "The making of a standard benchmark for open system storage," *J. Comput. Resource Management*, no. 101, pp. 26–32, Winter 2001.
- [18] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-volatile memory for fast, reliable file systems," in *Operating Systems Review*, vol. 26, pp. 10–22, October 1992.
- [19] P. Biswas, K. Ramakrishnan, D. Towsley, and C. Krishna, "Performance analysis of distributed file systems with non-volatile caches," in *Proc. 2nd Int. Symp. High Perf. Distributed Computing*, pp. 252–262, 1993.
- [20] W. W. Hsu, A. J. Smith, and H. C. Young, "I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level," *ACM Trans. Database Syst.*, vol. 26, no. 1, pp. 96–143, 2001.
- [21] L. Huang and T. Chiueh, "Experiences in building a software-based SATF scheduler," Tech. Rep. ECSL-TR81, SUNY at Stony Brook, July 2001.
- [22] F. J. Corbató, "A paging experiment with the multics system," in *In Honor of P. M. Morse*, pp. 217–228, MIT Press, 1969. Also as MIT Project MAC Report MAC-M-384, May 1968.
- [23] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
- [24] M. Seltzer, P. Chen, and J. Ousterhout, "Disk scheduling revisited," in *Proc. USENIX Winter Tech. Conf.*, pp. 313–324, 1990.
- [25] D. M. Jacobson and J. Wilkes, "Disk scheduling algorithms based on rotational position," tech. rep., HPL-CSP-91-7, HP Labs, Mar 1991.
- [26] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," in *SIGMETRICS*, pp. 241–251, 1994.
- [27] E. G. Coffman, L. A. Klimko, and B. Ryan, "Analysis of scanning policies for reducing disk seek times," *SIAM J. Comput.*, vol. 1, no. 3, pp. 269–279, 1972.
- [28] P. J. Denning, "Effects of scheduling on file memory operations," in *Proc. AFIPS Spring Joint Comput. Conf.*, pp. 9–21, 1967.
- [29] P. H. Seaman, R. A. Lind, and T. L. Wilson, "An analysis of auxiliary-storage activity," *IBM Systems Journal*, vol. 5, no. 3, pp. 158–170, 1966.
- [30] A. G. Merten, *Some quantitative techniques for file organization*. PhD thesis, University of Wisconsin, 1970.
- [31] R. Geist and S. Daniel, "A continuum of disk scheduling algorithms," *ACM Trans. Comput. Syst.*, vol. 5, no. 1, pp. 77–92, 1987.
- [32] T. R. Haining, *Non-volatile Cache Management For Improving Write Response Time with Rotating Magnetic Media*. PhD thesis, Ph.D. Dissertation, University of California, Santa Cruz, 2000.
- [33] J.-F. Paris, T. R. Haining, and D. D. E. Long, "A stack model based replacement policy for a non-volatile cache," in *Proc. IEEE Sym. Mass Storage Sys.*, pp. 217–224, March 2000.

Secure Deletion for a Versioning File System

Zachary N. J. Peterson Randal Burns Joe Herring
Adam Stubblefield Aviel D. Rubin
The Johns Hopkins University, Baltimore, MD

Abstract

We present algorithms and an architecture for the secure deletion of individual versions of a file. The principal application of this technology is federally compliant storage; it is designed to eliminate data after a mandatory retention period. However, it applies to any storage system that shares data between files, most notably versioning file systems. We compare two methods for secure deletion that use a combination of authenticated encryption and secure overwriting. We also discuss implementation issues, such as the demands that secure deletion places on version creation and the composition of file system metadata. Results show that new secure deletion techniques perform orders of magnitude better than previous methods.

1 Introduction

Versioning storage systems are increasingly important in research and commercial applications. Versioning has been recently identified by Congress as mandatory for the maintenance of electronic records of publicly traded companies (Sarbanes-Oxley, Gramm-Leach-Bliley), patient medical records (HIPAA), and federal systems (FISMA).

Existing versioning storage systems overlook fine-grained, secure deletion as an essential requirement. Secure deletion is the act of removing digital information from a storage system so that it can never be recovered. Fine-grained refers to removing individual files or versions of a file, while preserving all other data in the system.

Secure deletion is valuable to security conscious users and organizations. It protects the privacy of user data and prevents the discovery of information on retired or sold computers. Traditional data deletion, or “emptying the trash”, simply frees blocks for allocation at a later time; the data persists, fully readable and intact. Even when

data are overwritten, information may be reconstructed using expensive forensic techniques, such as magnetic force microscopy [42].

We are particularly interested in using secure deletion to limit liability in the regulatory environment. By securely deleting data after they have fallen out of regulatory scope, *e.g.* seven years for corporate records in Sarbanes-Oxley, data cannot be recovered even if disk drives are produced and encryption keys revealed. Data are gone forever and corporations are not subject to exposure via subpoena or malicious attack.

Currently, there are no efficient methods for fine-grained secure deletion in storage systems that share data among files, such as versioning file systems [12, 20, 27, 25, 32, 35] and content-indexing systems [2, 26, 28].

The preferred and accepted methods for secure deletion in non-data sharing systems include: repeatedly overwriting data, such that the original data may not be recovered [17]; and, encrypting a file with a key and securely disposing of the key to make the data unrecoverable [8].

Block sharing hinders key management in encrypting systems that use key disposal. If a system were to use an encryption key per version, the key could not be discarded, as it is needed to decrypt shared blocks in future versions that share the encrypted data. To realize fine-grained secure deletion by key disposal, a system must keep a key for every shared block, resulting in an onerous number of keys that quickly becomes unmanageable. Fewer keys allow for more flexible security policies [22].

Secure overwriting also has performance concerns in versioning systems. In order to limit storage overhead, versioning systems often share blocks of data between file versions. Securely overwriting a shared block in a past version could erase it from subsequent versions. To address this, a system would need to detect data sharing dependencies among all versions before committing to a deletion. Also, in order for secure overwriting to be efficient, the data to be removed should be contiguous on

disk. Non-contiguous data blocks require many seeks by the disk head – the most costly disk drive operation. By their very nature, versioning systems are unable to keep the blocks of a file contiguous in all versions.

Our contributions include two methods for the secure deletion of individual versions that minimize the amount of secure overwriting while providing authenticated encryption. Our techniques combine disk encryption with secure overwriting so that a large amount of file data (any block size) are deleted by overwriting a small *stub* of 128 bits. We collect and store stubs contiguously in a file system block so that overwriting a 4K block of stubs deletes the corresponding 1MB of file data, even when file data are non-contiguous. Unlike encryption keys, stubs are not secret and may be stored on disk. Our methods do not complicate key management. We also present a method for securely deleting data out-of-band, a construct that lends itself to multiple parties with a shared interest in a single piece of data and to off-site back-ups.

To our knowledge, we are the first file system to adopt authenticated encryption (AE) [4], which provides both privacy and authenticity. Authenticity is essential to ensure that the data have not changed between being written to disk and read back. Particularly in environments where storage is virtualized or distributed and, thus, difficult to physically secure. Authenticated encryption requires message expansion – ciphertext are larger than the plaintext – which is an obstacle to its adoption. Encrypting file systems have traditionally used block ciphers, which preserve message size, to meet the alignment and capacity constraints of disk drives [5, 40, 22]. In practice, additional storage must be found for the expanded bits of the message. Our architecture creates a parallel structure to the inode block map for the storage of expanded bits of the ciphertext and leverages this structure to achieve secure deletion. Message expansion is fundamental to our deletion model.

We have implemented secure deletion and authenticated encryption in the ext3cow versioning file system, designed for version management in the regulatory environment [27]. Experimental results show that our methods for secure deletion improve deletion performance by several orders of magnitude. Also, they show that metadata maintenance and cryptography degrade file system performance minimally.

2 Related Work

Secure Deletion

Garfinkel and Shalat [16] survey methods to destroy digital data. They identify secure deletion as a serious and pressing problem in a society that has a high turnover in technology. They cite an increase in lawsuits and news reports on unauthorized disclosures, which they at-

tribute to a poor understanding of data longevity and a lack of secure deletion tools. They identify two methods of secure deletion that leave disk drives in a usable condition: secure overwriting and encryption.

In secure overwriting, new data are written over old data so that the old data are irrecoverable. Gutmann [17] gives a technique that takes 35 synchronous passes over the data in order to degauss the magnetic media, making the data safe from magnetic force microscopy. (Fewer passes may be adequate [16]). This technique has been implemented in user-space tools and in a Linux file system [3]. Secure overwriting has also been applied in the semantically-smart disk system [34].

For file systems that encrypt data on disk, data may be securely deleted by “forgetting” the corresponding encryption key [8]; without a key, data may never be decrypted and read again. This method works in systems that maintain an encryption key per file and do not share data between multiple files. The actual disposal of the encryption key may involve secure overwriting.

There are many user-space tools for secure deletion, such as *wipe*, *eraser*, and *bootandnuke*. These tools provide some protection when securely deleting data. However, they may leak information because they are unable to delete metadata. They may also leak data when the system truncates files. Further, they are difficult to use synchronously because they cannot be interposed between file operations.

The importance of deleting data has been addressed in other system components. A concept related to stub deletion has been used in memory systems [13], which erase a large segment of memory by destroying a small non-volatile segment. Securely deallocating memory limits the exposure of sensitive data [11]. Similar problems have been addressed by Gutmann [18, 19] and Viega [37].

Secure Systems

CFS [5] was an early effort that added encryption to a file system. In this user-space tool, local and remote (via NFS) encrypted directories are accessed via a separate mount point. All file data and metadata in that directory are encrypted using a pre-defined user key and encryption algorithm. CFS does not provide authenticated encryption.

NCryptfs [40] is a cryptographic file system implemented as a stackable layer in FiST [41]. The system is designed to be customizable and flexible for its users by providing many options for encryption algorithms and key requirements. It does not provide authenticated encryption.

Cryptoloop uses the Linux cryptographic API [24] and the loopback interface to provide encryption for blocks as they are passed through to the disk. While easy to ad-

minister for a single-user machine, cryptographic loop-back devices do not scale well to multi-user systems.

Our implementation of encryption follows the design of the CryptoGraphic Disk Driver (CGD) [15]. CGD replaces the native disk device driver with one that encrypts blocks as they are transferred to disk.

The encryption and storage of keys in the random-key encryption scheme resembles lock-boxes in the Plutus file system [22] in which individual file keys are stored in lock-boxes and sealed with a user's key.

Cryptography

Secure deletion builds upon cryptographic constructs that we adapt to meet the demands of a versioning file system. The principal methods that we employ are the all-or-nothing transform [29], secret-sharing [33], and authenticated encryption [4]. Descriptions of their operation and application appear in the appropriate technical sections.

3 Secure Deletion with Versions

We have developed an approach to secure deletion for versioning systems that minimizes the amount of secure overwriting, eliminates the need for data block contiguity, and does not increase the complexity of key management.

Secure deletion with versions builds upon authenticated encryption of data on disk. We use a keyed transform:

$$f_k(B_i, N) \rightarrow C_i || s_i$$

that takes a data block (B_i), a key (k) and a nonce (N) and creates an output that can be partitioned into an encrypted data block (C_i), where $|B_i| = |C_i|$, and a short *stub* (s_i), whose length is a parameter of the scheme's security. When the key (k) remains private, the transform acts as an authenticated encryption algorithm. To securely delete an entire block, only the stub needs to be securely overwritten. This holds *even if the adversary is later given the key (k)*, which models the situation in which a key is exposed, *e.g.* by subpoena. The stub reveals nothing about the key or the data, and, thus, stubs may be stored on the same disk. It may be possible to recover securely deleted data after the key has been exposed by a brute-force search for the stub. However, this is no easier than a brute-force search for a secret key and is considered intractable.

A distinct advantage of our file system architecture is the use of authenticated encryption [4]. Authenticated encryption is a transform by which data are kept both private *and* authentic. Many popular encryption algorithms, such as AES, by themselves, provide only privacy; they cannot guarantee that the decrypted plaintext is the same

as the original plaintext. When decrypting, an authenticated encryption scheme will take a ciphertext and return either the plaintext or an indication the ciphertext is invalid or unauthentic. A common technique for authenticated encryption is to combine a message authentication code (MAC) with a standard block cipher [4]. However, single pass methods exist [30].

Authenticated encryption is a feature not provided by encrypting file systems to date. This is because authenticated encryption algorithms expand data when encrypting; the resulting cipherblock is larger than the original plaintext. This causes a mismatch in the block and page size. File systems present a page of plaintext to the memory system, which fills completely a number of sectors on the underlying disk. The AE encrypted ciphertext is larger than and does not align with the underlying sectors. (Other solutions based on a file system or disk redesign are possible). Expansion results in a loss of transparency for the encryption system. We address the problem of data expansion and leverage the expansion to achieve secure deletion.

Our architecture for secure deletion with stubs does not complicate key management. It employs the same key-management framework used by disk-encrypting file systems based on block ciphers, such as Plutus [22] and NCryptfs [40]. It augments these to support authenticated encryption and secure deletion.

We present and compare two implementations of the keyed transform (f_k): one inspired by the all-or-nothing transform and the other based on randomized keys. Both algorithms allow for the efficient secure deletion of a single version. We also present extensions, based on secret-sharing, that allow for the out-of-band deletion of data by multiple parties.

3.1 AON Secure Deletion

The all-or-nothing (AON) transform is a cryptographic function that, given a partial output, reveals nothing about its input. No single message of a ciphertext can be decrypted in isolation without decrypting the entire ciphertext. The transform requires no additional keys. The original intention, as proposed by Rivest [29], was to prevent brute-force key search attacks by requiring the attacker to decrypt an entire message for each key guess, multiplying the work by a factor of the number of blocks in the message. Boyko presented a formal definition for the AON transform [9] and showed that the OAEP [4] scheme used in many Internet protocol standards meets his definition. AON has been proposed to make efficient smart-card transactions [6, 7, 21], message authentication [14], and threshold-type cryptosystems using symmetric primitives [1].

The AON transform is the most natural construct for

Input: Data Block d_1, \dots, d_m , Block ID id , Counter x , Encryption key K , MAC key M

```

1:  $ctr_1 \leftarrow id || x || 1 || 0^{128-|x|-|id|-1}$ 
2:  $c_1, \dots, c_m \leftarrow \text{AES-CTR}_K^{ctr_1}(d_1, \dots, d_m)$ 
3:  $t \leftarrow \text{HMAC-SHA-1}_M(c_1, \dots, c_m)$ 
4:  $ctr_2 \leftarrow id || x || 0 || 0^{128-|x|-|id|-1}$ 
5:  $x_1, \dots, x_m \leftarrow \text{AES-CTR}_t^{ctr_2}(c_1, \dots, c_m)$ 
6:  $x_0 \leftarrow x_1 \oplus \dots \oplus x_m \oplus t$ 
Output: Stub  $x_0$ , Ciphertext  $x_1, \dots, x_m$ 

```

(a) AON encryption

Input: Stub x_0 , Ciphertext x_1, \dots, x_m , Block ID id , Counter x , Encryption key K , MAC key M

```

1:  $ctr_2 \leftarrow id || x || 0 || 0^{128-|x|-|id|-1}$ 
2:  $t \leftarrow x_0 \oplus \dots \oplus x_m$ 
3:  $c_1, \dots, c_m \leftarrow \text{AES-CTR}_t^{ctr_2}(x_1, \dots, x_m)$ 
4:  $t' \leftarrow \text{HMAC-SHA-1}_M(c_1, \dots, c_m)$ 
5: if  $t' \neq t$  return  $\perp$ 
6:  $ctr_1 \leftarrow id || x || 1 || 0^{128-|x|-|id|-1}$ 
7:  $d_1, \dots, d_m \leftarrow \text{AES-CTR}_K^{ctr_1}(c_1, \dots, c_m)$ 
Output: Data Block  $d_1, \dots, d_m$ 

```

(b) AON decryption

Figure 1: Authenticated encryption and secure deletion for a single data block in a versioning file system using the all-or-nothing scheme.

the secure deletion of versions. We aim to minimize the amount of secure overwriting. We also aim to not complicate key management. AON fulfills both requirements while conforming to our deletion model. The all-or-nothing property of the transform allows the system to overwrite any small subset of a data block to delete the entire block; without all subsets, the block cannot be read. When combined with authenticated encryption, the AON transform creates a message expansion that is bound to the same all-or-nothing property. This expansion is the stub and can be securely overwritten to securely delete a block. Because the AON transform requires no additional keys, key management is no more complicated than a system that uses a block cipher.

We present our AON algorithm for secure deletion in Figure 1. The encryption algorithm (Figure 1(a)) takes as inputs: a single file system data block segmented into 128-bit plaintext messages (d_1, \dots, d_m), a unique identifier for the block (id), a unique global counter (x), an encryption key (K) and a MAC key (M). To encrypt, the algorithm first generates a unique encryption counter (ctr_1) by concatenating the block identifier (id) with the global counter (x) and padding with zeros (Step 1). When AES is in counter mode (AES-CTR), a counter is encrypted, and used as an initialization vector (IV) to the block cipher to prevent similar plaintext blocks encrypting to the same cipher block. The same counter and key combination should not be used more than once, so we use a block's physical disk address for id and the epoch in which it was written for x ; both characteristics exist within an inode and, by policy, are non-repeatable in a file system. An AES encryption of the data is performed in counter mode (AES-CTR) using a single file key (K) and the counter generated in Step 1 (ctr_1). This results in encrypted data (c_1, \dots, c_m). The encrypted data are authenticated (Step 3) using SHA-1 and MAC key (M) as

a keyed-hash for message authentication codes (HMAC). The authenticator (t) is then used as the key to re-encrypt the data (Step 5). The authenticator can be used in this manner because the output of SHA-1 is assumed to be random. A second counter (ctr_2) is used to prevent repetitive encryption. A stub (x_0) is generated (Step 6) by XOR-ing all the ciphertext message blocks (x_1, \dots, x_m) with the authenticator (t). The resulting stub is not secret, rather, it is an expansion of the encrypted data and is subject to the all-or-nothing property. The ciphertext (x_1, \dots, x_m) is written to disk as data, and the stub (x_0) is stored as metadata.

Decryption (Figure 1(b)) works similarly, but in reverse. The algorithm is given as inputs: the stub (x_0), the AON encrypted data block (x_1, \dots, x_m), the same block ID (id) and counter (x) as in the encryption, and the same encryption (K) and MAC (M) keys used to encrypt. The unique counter (ctr_2) is reconstructed (Step 1), the authenticator (t) is reconstructed (Step 2) and then used in the first round of decrypting the data (Step 3). An HMAC is performed on the resulting ciphertext (Step 4) and the result (t') is compared with the reconstructed authenticator (t) (Step 5). If the authenticators do not match, the data are not the same as when they were written. Lastly, the data are decrypted (Step 7), resulting in the original plaintext.

Despite the virtues of providing authenticated encryption with low performance and storage overheads, this construction of AON encryption suffers from a guessed-plaintext attack. After an encryption key has been revealed, if an attacker can guess the exact contents of a block of data, she can verify that the data were once in the file system. This attack does not reveal encrypted data. Once the key is disclosed, the attacker has all of the inputs to the encryption algorithm and may reproduce the ciphertext. The ciphertext may be compared to

Input: Data Block d_1, \dots, d_m , Block ID id , Counter x , Encryption key K , MAC key M

```

1:  $k \xleftarrow{R} \mathcal{K}_{AE}$ 
2:  $nonce \leftarrow id || x$ 
3:  $c_1, \dots, c_n \leftarrow \text{AE}_k^{nonce}(d_1, \dots, d_m)$ 
4:  $ctr \leftarrow id || x || 0^{128-|x|-|id|}$ 
5:  $c_0 \leftarrow \text{AES-CTR}_K^{ctr}(k)$ 
6:  $t \leftarrow \text{HMAC-SHA-1}_M(ctr, c_0)$ 
Output: Stub  $c_0, t, c_{m+1}, \dots, c_n$ , Ciphertext  $c_1, \dots, c_m$ 

```

(a) Random-key encryption

Input: Stub $c_0, t, c_{n+1}, \dots, c_m$, Ciphertext c_1, \dots, c_n , Block ID id , Counter x , Encryption key K , MAC key M

```

1:  $ctr \leftarrow id || x || 0^{128-|x|-|id|}$ 
2:  $t' \leftarrow \text{HMAC-SHA-1}_M(ctr, c_0, r)$ 
3: if  $t' \neq t$  return  $\perp$ 
4:  $k \leftarrow \text{AES-CTR}_K^{ctr}(c_0)$ 
5:  $nonce \leftarrow id || x$ 
6:  $d_1, \dots, d_n = \text{AE}_k^{nonce}(c_1, \dots, c_m)$ 
Output: Data Block  $d_1, \dots, d_n$ 

```

(b) Random-key decryption

Figure 2: Authenticated encryption and secure deletion for a single data block in a versioning file system using the random-key scheme.

the undeleted block of data, minus the deleted stub, to prove the existence of the data.

Such an attack is reasonable within the threat model of regulatory storage; a key may be subpoenaed in order to show that the file system contained specific data at some time. For example, to show that an individual had read and subsequently made attempts to destroy an incriminating email. This threat can be eliminated by adding a random postfix to the data block, though this will increase the size of the stub.

3.2 Secure Deletion Based on Randomized Keys

As mentioned by Rivest [29], avoiding such a text-guessing attack requires that an AON transform employ randomization so that the encryption process is not repeatable given the same inputs. The subsequent security construct generates a random key on a per-block basis.

Random-key encryption is not an all-or-nothing transform. Instead, it is a refinement of the Boneh key disposal technique [8]. Each data block is encrypted using a randomly generated key. When this randomly generated key is encrypted with the file key, it acts as a stub. Like AON encryption, random-key encryption makes use of authenticated encryption, minimizes the amount of data needed to be securely overwritten, and does not require the management of additional keys.

We give an algorithm for random-key secure deletion in Figure 2. To encrypt (Figure 2(a)), the scheme generates a random key, k , (Step 1) that is used to authenticate and encrypt a data block. Similar to the unique counters in the AON scheme, a unique nonce is generated (Step 2). Data is then encrypted and authenticated (Step 3), resulting in an expanded message. The algorithm is built upon any authenticated encryption (AE) scheme; AES and SHA-1 satisfy standard security def-

initions. To avoid the complexities of key distribution, we employ a single encryption (K) and MAC (M) key per file (the same keys as used in AON encryption) and use these keys to encrypt and authenticate the random key (k) (Step 5). The encrypted randomly-generated key (c_0) serves as the stub. The expansion created by the AE scheme in Step 3 (c_{m+1}, \dots, c_n), and the authentication of the encrypted random key (t) does not need to be securely overwritten to permanently destroy data.

An advantage of random-key encryption over AON encryption is its speed. For example, when the underlying AE is OCB [30], only one pass over the data is made and it is fully parallelizable. However, the algorithm suffers from a larger message expansion: 384 bits per disk block are required instead of 128 required for the AON scheme. We are exploring other more space-efficient algorithms. We have developed another algorithm that requires no more bits than the underlying AE scheme. Unfortunately, this is based on OAEP and a Luby-Rackoff construction [23] and is only useful for demonstrating that space efficient constructions do exist. It is far too slow to be used in practice, requiring six expensive passes over the data.

3.3 Other Secure Deletion Models

Our secure deletion architecture was optimized for the most common deletion operation: deleting a single version. However, there are different models for removing data that may be more efficient in certain circumstances. These include efficiently removing a block or all blocks from an entire version chain and securely deleting data shared by multiple parties.

3.3.1 Deleting a Version Chain

AON encryption allows for the deletion of a block of data from an entire version chain. Due to the all-or-nothing properties of the transform, the secure overwriting of *any* 128 bits of a data block results in the secure deletion of that block, even if the stub persists. When a user wishes to delete an entire version chain, *i.e.* all blocks associated with all versions of a file, it may be more efficient to securely overwrite the blocks themselves rather than each version's stubs. This is because overwriting is slow and many blocks are shared between versions. For example, to delete a large log file to which data has only been appended, securely deleting all the blocks in the most recent version will delete all past versions. Ext3cow provides a separate interface for securely deleting data blocks from all versions. If a deleted block was shared, it is no longer accessible to other versions, despite their possession of the stub.

Randomized-key encryption does not hold this advantage; only selective components may be deleted, *i.e.* c_0 . Thus, in order to delete a block from all versions, the system must securely overwrite all stub occurrences in a version chain, as opposed to securely overwriting only 128 bits of a data block in an AON scheme. To remedy this, a key share (Section 3.3.2) could be stored alongside the encrypted data block. When the key share is securely overwritten, the encrypted data are no longer accessible in any version. However, this strategy is not practical in most file systems, owing to block size and alignment constraints. Storage for the key share must be provided and there is no space in the file system block. The shares could be stored elsewhere, as we have with deletion stubs, but need to be maintained on a per-file, rather than per-version, basis.

3.3.2 Secure Deletion with Secret-Sharing

The same data are often stored in more than one place. An obvious example of this are remote back-ups. It is desirable that when data fall out of regulatory scope, all copies of data are destroyed. Secret-sharing provides a solution.

Our random-key encryption scheme allows for the separation of the randomly-generated encryption key into n key shares. This is a form of an (m, n) secret-sharing scheme [33]. In secret-sharing, Shamir shows how to divide data into n shares, such that any m shares can reconstruct the data, but where $m - 1$ shares reveals nothing about the data. We are able to compose a single randomly generate encryption key (k) from multiple key shares. An individual key share may then be given to a user with an interest in the data, distributing the means to delete data. If a single key share is independently deleted, the corresponding data are securely deleted and

the remaining key shares are useless. Without all key shares, the randomly generated encryption key may not be reconstructed and decryption will fail.

Any number of randomly generated keys may be created in Step 1 (Figure 2(a)) and composed to create a single encryption key (k). To create two key shares (a $(2, 2)$ scheme), Step 1 may be replaced with:

$$\begin{aligned}\ell, r &\xleftarrow{R} \mathcal{K}_{AE} \\ k &\leftarrow \ell \oplus r\end{aligned}$$

The stub (c_0) then becomes the encryption of any one key share, for example:

$$c_0 \leftarrow \text{AES-CTR}_K^{\text{ctr}}(\ell)$$

With an (n, n) key share scheme, any single share may be destroyed to securely delete the corresponding data. The caveat being that all key shares must be present at the time of decryption. This benefits parties who have a shared interest in the same data. For example, a patient may hold a key share for their medical records on a smartcard, enabling them to control access to their records and also independently destroy their records without access to the storage system.

This feature extends to the management of securely deleting data from back-ups systems. Data stored at an off-site location may be deleted out-of-band by overwriting the appropriate key shares. In comparison, without secret-sharing, all copies of data would need to be collected and deleted to ensure eradication. Once data are copied out of the secure deletion environment, no assurance as to the destruction of the data may be made.

3.4 Security Properties

Confidence is gained in modern cryptographic constructions through the use of reductionist arguments: it is shown that if an adversary can break a particular construction, he can also break the underlying primitives that are employed. For example, AES in CTR mode can be shown to be secure so long as the AES algorithm is itself secure.

As was previously pointed out, the authenticated AON scheme is not secure as it falls victim to a plaintext guessing attack. Even if this particular problem is fixed (by appending a random block to the plaintext, thereby increasing the size of the stub), the construction is not necessarily secure. Due to some technical problems with the model, a proof that this type of “package transform” construction reduces to the security of the underlying block cipher has eluded cryptographers for several years.

The random keys construction is provably secure (under reasonable definitions for this application) so long

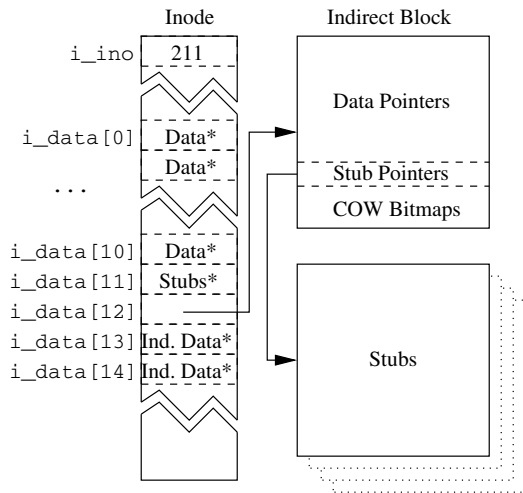


Figure 3: Metadata architecture to support stubs.

as the underlying authenticated encryption scheme is secure, AES is secure, HMAC-SHA1 is secure, and SHA-1 acts as a random oracle. We omit the formal definitions and proofs from this work.

4 Architecture

We have implemented secure deletion in ext3cow [27], an open-source, block-versioning file system designed to meet the requirements of electronic record management legislation. Ext3cow supports file system snapshot, per-file versioning, and a time-shifting interface that provides real-time access to past versions. Versions of a file are implemented by chaining inodes together where each inode represents a version of a file.

4.1 Metadata for Secure Deletion

Metadata in ext3cow have been retrofitted to support versioning and secure deletion. For versioning, ext3cow employs a copy-on-write policy when writing data. Instead of overwriting old data with new data, ext3cow allocates a new disk block in which to write the new data. A new inode is created to record the modification and is chained to the previous inode. Each inode represents a single version and, as a chain, symbolizes the entire version history of a file. To support versioning, ext3cow “steals” address blocks from an inode’s indirect blocks to embed bitmaps used to manage copy-on-written blocks. In a 4K indirect block (respectively, doubly or triply indirect blocks), the last thirty-two 32-bit words of the block contain a bitmap with a bit for every block referenced in that indirect block.

A similar “block stealing” design was chosen for managing stubs. A number of block addresses in the inode

and the indirect blocks have been reserved to point to blocks of stubs. Figure 3 illustrates the metadata architecture. The number of direct blocks in an inode has been reduced by one, from twelve to eleven, for storage of stubs ($i_data[11]$) that correspond to the direct blocks. Ext3cow reserves words in indirect blocks to be used as pointers to blocks of stubs.

The number of stub block pointers depends on the file system block size and the encryption method. In AON encryption, four stub blocks are required to hold the stubs corresponding to the 4MB of data described by a 4K indirect block. Because of the message expansion and authentication components of the randomized-key scheme (c_{n+1}, \dots, c_m, t), sixteen stub blocks must be reserved; four for the deletable stubs and twelve for the expansion and authentication. Only the stub blocks must be securely overwritten in order to permanently delete data.

All stub blocks in an indirect block are allocated with strict contiguity. This has two benefits: when securely deleting a file, contiguous stub blocks may be securely overwritten together, improving the time to overwrite. Second, stub blocks may be more easily read when performing an I/O. Stub blocks should not increase the number of I/Os performed by the drive for a read. Ext3cow makes efforts to co-locate data, metadata and stub blocks in a single disk drive track, enabling all to be read in single I/O.

Because the extra metadata borrows space from indirect blocks, the design reduces the maximum file size. The loss is about 16%. With a 4K block size, ext3cow represents files up to 9.03×10^8 blocks in comparison to 1.07×10^9 blocks in ext3. The upcoming adoption of quadruply indirect blocks by ext3 [36] will remove practical file size limitations.

4.2 The Secure Block Device Driver

All encryption functionality is contained in a secure block device driver. By encapsulating encryption in a single device driver, algorithms are modular and independent of the file system or other system components. This enables any file system that supports the management of stubs to utilize our device driver.

When encrypting, a data page is passed to the device driver. The driver copies the page into its private memory space, ensuring the user’s image of the data is not encrypted. The driver encrypts the private data page, generates a stub, and passes the encrypted page to the low level disk driver. The secure device driver interacts with the file system twice: once to acquire encryption and authentication keys and once to write back the generated stub.

Cryptography in the device driver was built upon the pre-existing cryptographic API available in the Linux

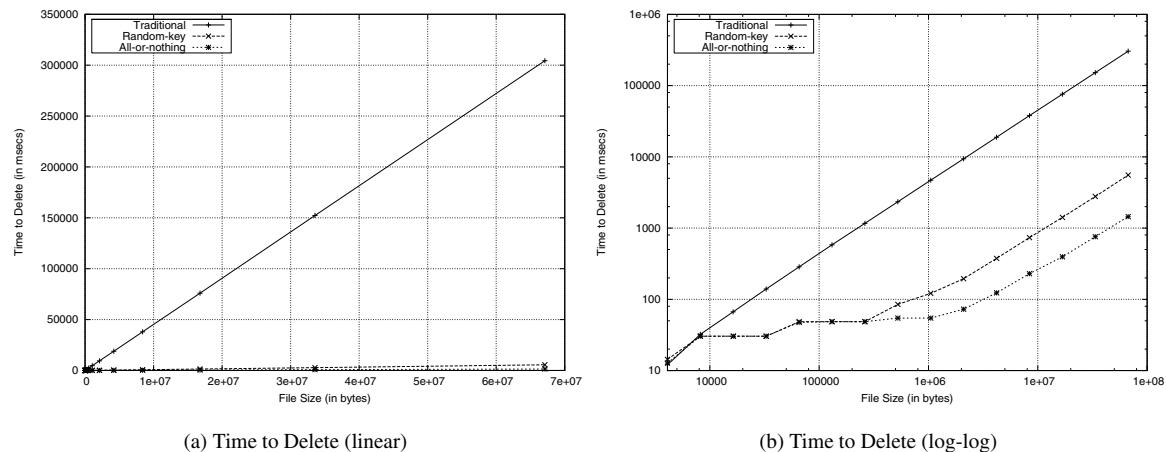


Figure 4: The time to securely delete files for the secure overwriting (traditional), all-or-nothing, and random-key techniques.

kernel [24], namely the AES and SHA-1 algorithms. Building upon existing constructs simplified development, and aids correctness. Further, it allows for the security algorithms to evolve, giving opportunity for the secure deletion transforms to be updated as more secure algorithms become available. For instance, the security of SHA-1 has been recently called into question [38].

We plan to release the secure device driver to the Linux community under an open source license via the ext3cow website, www.ext3cow.com.

4.3 Security Policies

When building an encrypting, versioning file system, certain policies must be observed to ensure correctness. In our security model, a stub may never be re-written in place once committed to disk. Violating this policy places new stub data over old stub data, allowing the old stub to be recoverable via magnetic force microscopy or other forensic techniques.

With secure deletion, I/O drives the creation of versions. Our architecture mandates a new version whenever a block and a stub are written to disk. Continuous versioning, *e.g.* CVFS [35], meets this requirement, because it creates a new version on every `write()` system call. However, for many users, continuous versioning may incur undesirable storage overheads, approximately 27% [27, 35]. Most systems create versions less frequently. As a matter of policy, *e.g.* daily, on every file open, *etc.*; or, explicitly through a snapshot interface.

The demands of secure deletion may be met without continuous versioning. Ext3cow reduces the creation of versions based on the observation that multiple writes to the same stub may be aggregated in memory prior to

reaching disk. We are developing write-back caching policies that delay writes to stub blocks and aggregate multiple writes to the same stub or writes to multiple stubs within the same disk sector. Stub blocks may be delayed even when the corresponding data blocks are written to disk; data may be re-written without security exposure. A small amount of non-volatile, erasable memory or an erasable journal would be helpful in delaying disk writes when the system call specifies a synchronous write.

5 Experimental Results

We measure the impact that AON and random-key secure deletion have on performance in a versioning file system. We begin by measuring the performance benefits of deletion achieved by AON and random-key secure deletion. We then use the Bonnie++ benchmark suite to stress the file system under different cryptographic configurations. Lastly, we explore the reasons why secure deletion is a difficult problem for versioning file systems through trace-driven file system aging experiments. All experiments were performed on a Pentium 4, 2.8GHz machine with 1GB of RAM. Bonnie++ was run a 80GB partition of a Seagate Barracuda ST380011A disk drive.

5.1 Time to Delete

To examine the performance benefits of our secure deletion techniques, we compared our all-or-nothing and random-key algorithms with Gutmann's traditional secure overwriting technique. Files, sized 2^n blocks for $n = 0, 1, \dots, 20$, were created; for 4KB blocks, this a

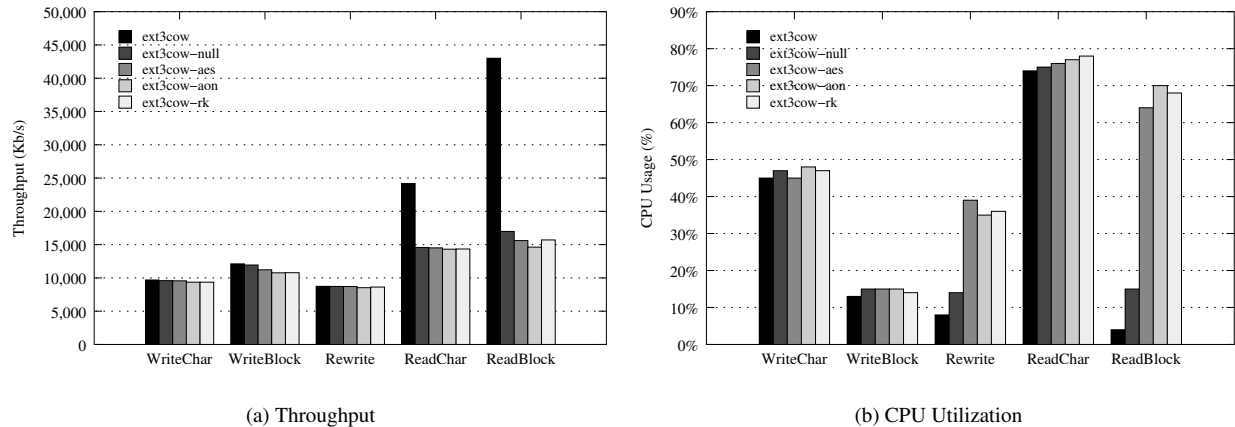


Figure 5: Bonnie++ throughput and CPU utilization results.

file size range of 4KB to 4GB. Each file was then securely deleted using each of the three secure deletion methods, and the time to do so was measured. Because no versioning is taking place, files are relatively contiguous on disk. Further, no blocks are shared between versions so all blocks of the file are overwritten.

Figure 4(a) demonstrates the dramatic savings in time that can be achieved by using stub deletion. Files between 2^{16} and 2^{20} were truncated for clarity. AON deletion beats traditional deletion by a factor of 200 for 67MB files (2^{15} blocks), with random-key deletion performing slightly worse than AON. Differences are better seen in Figure 4(b), a log-log plot of the same result.

AON and random-key deletion perform similarly on files allocated only with direct blocks (between 2^0 and approximately 2^4 blocks), and begin to diverge at 2^7 blocks. By the time files are allocated using doubly indirect blocks (between 2^9 and 2^{10} blocks) the performance of random-key and AON differ substantially. This is due to the larger stub size needed for random-key deletion, requiring more secure overwriting of stub blocks.

5.2 Bonnie++

Bonnie++ is a well-known performance benchmark that quantifies five aspects of file system performance based on observed I/O bottlenecks in a UNIX-based file system. Bonnie++ performs I/O on large files (for our experiment, two 1-GB files) to ensure I/O requests are not served out of the disk's cache. For each test, Bonnie++ reports throughput, measured in kilobytes per second, and CPU utilization, as a percentage of CPU usage. Five operations are tested: (1) each file is written sequentially by character, (2) each file is written sequentially by block, (3) the files are sequentially read and rewritten, (4) the files are read sequentially by charac-

ter, and (5) the files are read sequentially by block. We compare the results of five file system modes: ext3cow, ext3cow-null, ext3cow-aes, ext3cow-aon and ext3cow-rk. Respectively, they are: a plain installation of ext3cow with no secure device driver. Ext3cow with a secure device driver that does no encryption. Ext3cow with a secure device driver that does a simple AES encryption. Ext3cow with a secure device driver that runs the all-or-nothing algorithm, and ext3cow with a secure device driver that runs the random-key algorithm. Ext3cow performs comparably with ext3 [27]. Results are the product of an average of 10 runs of Bonnie++ on the same partition.

Figure 5(a) presents throughput results for each Bonnie++ test. When writing, throughput suffers very little in the presence of cryptography. The largest difference occurs when writing data a block at a time; AON encryption reduces throughput by 1.3 MB/s, from 12.1 MB/s to 10.8 MB/s. This result is consistent with the literature [39]. A more significant penalty is incurred when reading. However, we believe this to be an artifact of the driver and not the cryptography, as the null driver (the secure device driver employing no cryptography) experiences the same performance deficit. The problem stems from the secure device driver's inability to aggregate local block requests into a single large request. We are currently implementing a request clustering algorithm that will eliminate the disparity. In the meantime, the differences in the results for the null device driver and device drivers that employ cryptography are minor: a maximum difference of 200 K/s for character reading and 1.2 MB/s for block reading. Further, the reading of stubs has no effect on the ultimate throughput. We attribute this to ext3cow's ability to co-locate stubs with the data they represent. Because it is based on ext3 [10], ext3cow employs block grouping to keep metadata and

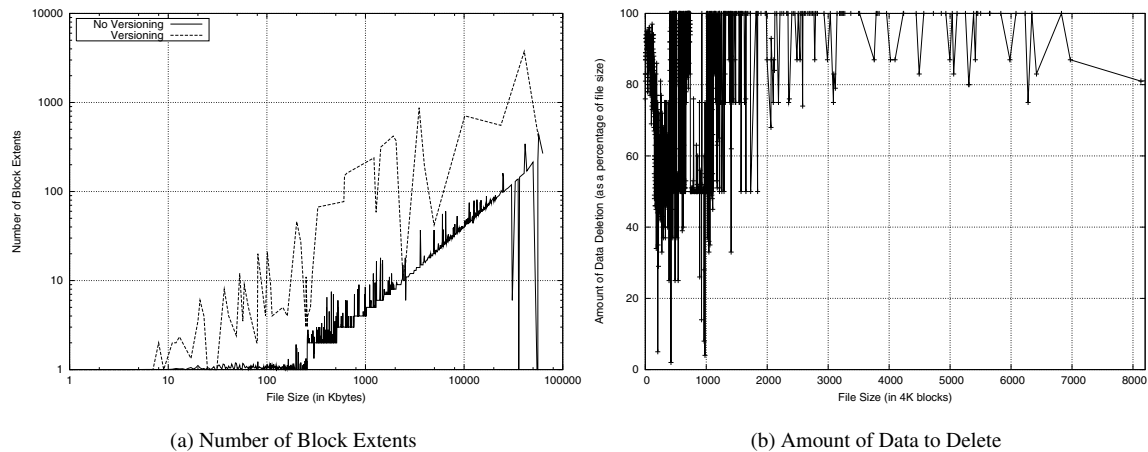


Figure 6: Results of trace-driven file system aging experiments.

data near each other on disk. Thus, track caching on disk and read-ahead in ext3cow put stubs into the disk and system cache, making them readily available when accessing the corresponding data.

To gauge the impact of file system cryptography on the CPU, we measured the CPU utilization for each Bonnie++ test. Results are presented in Figure 5(b). When writing, cryptography, as a percentage of the total CPU, has nearly no effect. This makes sense, as more of the CPU is utilized by the operating system for writing than for reading. Writes may perform multiple memory copies, allocate memory pages, and update metadata. Similarly, reading data character by character is also CPU intensive, due to buffer copying and other memory management operations, so cryptography has a negligible effect. Cryptography does have a noticeable effect when reading data a block at a time, evident in the rewrite and block read experiments. Because blocks match the page size in ext3cow, little time must be spent by the CPU to manage memory. Thus, a larger portion of CPU cycles are spent on decryption. However, during decryption, the system remains I/O bound, as the CPU never reaches capacity. These results are consistent with recent findings [39] that the overheads of cryptography are acceptable in modern file systems.

The cost of cryptography for secure deletion does not outweigh the penalties for falling out of regulatory compliance. In the face of liability for large scale identity theft, the high cost of litigation, and potentially ruinous regulatory penalties, cryptography should be considered a relatively low cost and necessary component of regulatory storage systems.

5.3 Trace-Driven Experiments

We present results that quantify the difficulty of achieving good performance when securely deleting data that have fallen out of regulatory scope. We replayed four months of file system call traces [31] on an 80G ext3cow partition, taking snapshots every second. This results in 4.2 gigabytes of data in 81674 files.

We first examine the amount of external fragmentation that results from versioning. External fragmentation is the phenomenon of file blocks in nonadjacent disk addresses. This causes multiple disk drive seeks to read or delete a file. Ext3cow uses a copy-on-write scheme to version files [27]. This precludes the file system from keeping all blocks of a version strictly contiguous. Because seeks are an expensive operations, fragmentation is detrimental to the performance of traditional secure overwriting. Figure 6(a) shows the effect versioning has on block fragmentation. Versioning increases dramatically the average number of block extents – regions of contiguous blocks. This is in comparison to the ext3 file system without versioning. Note the log-log scale. Some files have as many as 1000 block extents. This is the result of files receiving lots copy-on-write versioning.

In practice, secure deletion provides more benefit than microbenchmark results would indicate (Section 5.1). Given that seeking is the most expensive disk operation, traditional secure overwriting scales with the number of block extents that need to be overwritten. For AON or random-key secure deletion, the number of extents depends only upon the file size, not the fragmentation of data. Deletion performance does not degrade with versioning. For secure overwriting of the file data, performance scales with the number of block extents. Given the large degree of fragmentation generated through ver-

sioning, isolating deletion performance from file contiguity is essential.

Despite the high degree of copy-on-write and fragmentation, trace results show that there are considerable data to delete in each version, *i.e.* deletion is non-trivial. When a version of a file falls out of scope, much of its data are unique to that version and, thus, need to be securely deleted. This is illustrated in Figure 6(b). This graph shows the average amount of data that needs to be deleted as a percentage of the file size. There are very few files that have fewer than 25% unique blocks. Most versions need 100% of their blocks deleted. This is not unexpected as many files are written once and never modified. This is much more important for larger files which are more sensitive to deletion performance; stub deletion offers less benefit when deleting very small files. Even the largest files in the file system contain mostly unique data.

6 Applicability to Other Data System

There is potential for the reuse of the AON and random-key algorithms for secure deletion in any storage system that shares data among files. Content-indexing systems, such as Venti [28], LBFS [26], and pStore [2], have the same deletion problems and our technology translates directly. Content-indexing stores a corpus of data blocks (for all files) and represents a file as an assemblage of blocks in the corpus. Files that share blocks in the corpus have the same dependencies as do copy-on-write versions.

7 Conclusions

We define a model for secure deletion in storage systems that share data between files, specifically, versioning file systems that comply with federal regulations. Our model supports authenticated encryption, a unique feature for file systems. A data block is encrypted and converted into a ciphertext block and a small stub. Securely overwriting the stub makes the corresponding block irrecoverable.

We present two algorithms within this model. The first algorithm employs the all-or-nothing transform so that securely overwriting a stub or any 128 bit block of a ciphertext securely deletes the corresponding disk block. The second algorithm generates a random key per block in order to make encryption non-repeatable. The first algorithm produces more compact stubs and supports a richer set of deletion primitives, whereas the second algorithm provides stronger privacy guarantees.

Both secure deletion algorithms meet our requirement of minimizing secure overwriting, resulting in a 200

times speed-up over previous techniques. The addition of stub metadata and a cryptographic device driver degrade performance minimally. We have implemented secure deletion in the ext3cow versioning file system for Linux and in a secure device driver. Both are open-source and available for download at the project's webpage.

8 Acknowledgments

This work was supported in part by NSF award CCF-0238305, by the DOE Office of Science award P020685, the IBM Corporation, and by NSF award IIS-0456027.

References

- [1] ANDERSON, R. The dancing bear – a new way of composing ciphers. In *Proceedings of the International Workshop on Security Protocols* (April 2004).
- [2] BATTEN, C., BARR, K., SARAF, A., AND TREPETIN, S. pStore: A secure peer-to-peer backup system. Technical Memo MIT-LCS-TM-632, Massachusetts Institute of Technology Laboratory for Computer Science, October 2002.
- [3] BAUER, S., AND PRIYANTHA, N. B. Secure data deletion for Linux file systems. In *Proceedings of the USENIX Security Symposium* (August 2001).
- [4] BELLARE, M., AND NAMPREMPRE, C. Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology - Asiacrypt'00 Proceedings* (2000), vol. 1976, Springer-Verlag. Lecture Notes in Computer Science.
- [5] BLAZE, M. A cryptographic file system for UNIX. In *Proceedings of the ACM conference on Computer and Communications Security* (November 1993), pp. 9–16.
- [6] BLAZE, M. High-bandwidth encryption with low-bandwidth smartcards. In *Fast Software Encryption* (1996), vol. 1039, pp. 33–40. Lecture Notes in Computer Science.
- [7] BLAZE, M., FEIGENBAUM, J., AND NAOR, M. A formal treatment of remotely keyed encryption. In *Advances in Cryptology – Eurocrypt'98 Proceedings* (1998), vol. 1403, pp. 251–265. Lecture Notes in Computer Science.
- [8] BONEH, D., AND LIPTON, R. A revocable backup system. In *Proceedings of the USENIX Security Symposium* (July 1996), pp. 91–96.
- [9] BOYKO, V. On the security properties of OAEP as an all-or-nothing transform. In *Advances in Cryptology - Crypto'99 Proceedings* (August 1999), Springer-Verlag, pp. 503–518. Lecture Notes in Computer Science.
- [10] CARD, R., TS'O, T. Y., AND TWEEDIE, S. Design and implementation of the second extended file system. In *Proceedings of the Amsterdam Linux Conference* (1994).
- [11] CHOW, J., PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the USENIX Security Symposium* (August 2005), pp. 331–346.
- [12] CORNELL, B., DINDA, P. A., AND BUSTAMANTE, F. E. Way-back: A user-level versioning file system for Linux. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2004), pp. 19–28.

- [13] CRESCENZO, G. D., FERGUSON, N., IMPAGLIAZZO, R., AND JAKOBSSON, M. How to forget a secret. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science* (1999), vol. 1563, Springer-Verlag, pp. 500–509. Lecture Notes in Computer Science.
- [14] DODIS, Y., AND AN, J. Concealment and its applications to authenticated encryption. In *Advances in Cryptology – Eurocrypt’03 Proceedings* (2003), vol. 2656. Lecture Notes in Computer Science.
- [15] DOWDESWELL, R., AND IOANNIDIS, J. The CryptoGraphic disk driver. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2003), pp. 179–186.
- [16] GARFINKEL, S. L., AND SHELAT, A. Remembrance of data passed: A study of disk sanitization practices. *IEEE Security and Privacy* 1, 1 (2003), 17–27.
- [17] GUTMANN, P. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the USENIX Security Symposium* (July 1996), pp. 77–90.
- [18] GUTMANN, P. Software generation of practically strong random numbers. In *Proceedings of the USENIX Security Symposium* (January 1998), pp. 243–257.
- [19] GUTMANN, P. Data remanence in semiconductor devices. In *Proceedings of the USENIX Security Symposium* (August 2001), pp. 39–54.
- [20] HITZ, D., LAU, J., AND MALCOM, M. File system design for an NFS file server appliance. In *Proceedings of the Winter USENIX Technical Conference* (January 1994), pp. 235–246.
- [21] JAKOBSSON, M., STERN, J., AND YUNG, M. Scramble all. Encrypt small. In *Fast Software Encryption* (1999), vol. 1636. Lecture Notes in Computer Science.
- [22] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 29–42.
- [23] LUBY, M., AND RACKOFF, C. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing* 17, 2 (April 1988), 373–386.
- [24] MORRIS, J. The Linux kernel cryptographic API. *Linux Journal*, 108 (April 2003).
- [25] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. A versatile and user-oriented versioning file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2004), pp. 115–128.
- [26] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (October 2001), pp. 174–187.
- [27] PETERSON, Z., AND BURNS, R. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage* 1, 2 (2005), 190–212.
- [28] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)* (January 2002), pp. 89–101.
- [29] RIVEST, R. L. All-or-nothing encryption and the package transform. In *Proceedings of the Fast Software Encryption Conference* (1997), vol. 1267, pp. 210–218. Lecture Notes in Computer Science.
- [30] ROGAWAY, P., BELLARE, M., BLACK, J., AND KROVET, T. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *Proceedings of the ACM Conference on Computer and Communications Security* (November 2001), pp. 196–205.
- [31] ROSELLI, D., AND ANDERSON, T. E. Characteristics of file system workloads. Research report, University of California, Berkeley, June 1996.
- [32] SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (December 1999), pp. 110–123.
- [33] SHAMIR, A. How to share a secret. *Communications of the ACM* 22, 11 (1979), 612–613.
- [34] SIVATHANU, M., BAIRAVASUNDATAM, L., ARPACI-DUSSAEU, A. C., AND ARPACI-DUSSEAU, R. H. Life or Death at Block-Level. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (December 2004), pp. 379–394.
- [35] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)* (March 2003), pp. 43–58.
- [36] TS’O, T. Y., AND TWEEDIE, S. Planned extensions to the Linux ext2/ext3 filesystem. In *Proceedings of the USENIX Technical Conference, FREENIX Track* (June 2002), pp. 235–243.
- [37] VIEGA, J., AND MCGRAW, G. *Building Secure Software*. Addison-Wesley, 2002.
- [38] WANG, X., YIN, Y. L., AND YU, H. Finding collisions in the full SHA-1. In *Advances in Cryptology - Crypto’05 Proceedings* (August 2005), Springer-Verlag. Lecture Notes in Computer Science. To appear.
- [39] WRIGHT, C., DAVE, J., AND ZADOK, E. Cryptographic file systems performance: What you don’t know can hurt you. In *Proceedings of the IEEE Security in Storage Workshop (SISW)* (October 2003), pp. 47–61.
- [40] WRIGHT, C. P., MARTINO, M., AND ZADOK, E. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the USENIX Technical Conference* (June 2003), pp. 197–210.
- [41] ZADOK, E., AND NIEH, J. FiST: A language for stackable file systems. In *Proceedings of the USENIX Technical Conference* (June 2000), pp. 55–70.
- [42] ZHU, J.-G., LUO, Y., AND DING, J. Magnetic force microscopy study of edge overwrite characteristics in thin film media. *IEEE Transaction on Magnetism* 30, 6 (1994), 4242–4244.

TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study

Jinpeng Wei and Calton Pu
Georgia Institute of Technology
{weijp,calton}@cc.gatech.edu

ABSTRACT

Due to their non-deterministic nature, Time of Check To Time of Use (TOCTTOU) vulnerabilities in Unix-style file systems (e.g., Linux) are difficult to find and prevent. We describe a comprehensive model of TOCTTOU vulnerabilities, enumerating 224 file system call pairs that may lead to successful TOCTTOU attacks. Based on this model, we built kernel monitoring tools that confirmed known vulnerabilities and discovered new ones (in often-used system utilities such as *rpm*, *vi*, and *emacs*). We evaluated the probability of successfully exploiting these newly discovered vulnerabilities and analyzed in detail the system events during such attacks. Our performance evaluation shows that the dynamic monitoring of system calls introduces non-negligible overhead in microbenchmark of those file system calls, but their impact on application benchmarks such as Andrew and PostMark is only a few percent.

Categories and Subject Descriptors

D.4.3: File Systems Management – *Access methods*;
D.4.5: Reliability – *verification*; D.4.6: Security and Protection – *Access controls*.

General Terms

Reliability, Experimentation, Security.

Keywords

Race detection

1 Introduction

TOCTTOU (Time Of Check To Time Of Use) is a well known security problem [1] in file systems with weak synchronization semantics (e.g., Unix file system). A TOCTTOU vulnerability requires two steps [2]. First, a vulnerable program checks for a file status. Second, the program operates on the file assuming the original file status remained invariant during execution. For example, *sendmail* may check for a specific attribute of a mailbox (e.g., it is not a symbolic link) in step one and then append new messages (as root) in step two. Because the two steps are not executed atomically, a local attacker (mailbox owner) can exploit the window of vulnerability between the two steps by deleting his/her

mailbox and replacing it with a symbolic link to */etc/passwd*. If the replacement is completed within the window and the new messages happen to be syntactically correct */etc/passwd* entries with root access, then *sendmail* may unintentionally give unauthorized root access to a normal user (the attacker).

TOCTTOU vulnerabilities are a very significant problem. For example, between 2000 and 2004, we found 20 CERT [14] advisories on TOCTTOU vulnerabilities. They cover a wide range of applications from system management tools (e.g., */bin/sh*, *shar*, *tripwire*) to user level applications (e.g., *gpm*, Netscape browser). A similar list compiled from BUGTRAQ [16] mailing list is shown in Table 1. The CERT advisories affected many operating systems, including: Caldera, Conectiva, Debian, FreeBSD, HP-UX, Immunix, MandrakeSoft, RedHat, Sun Solaris, and SuSE. In 11 of the CERT advisories, the attacker was able to gain unauthorized root access. TOCTTOU vulnerabilities are widespread and cause serious consequences.

Table 1: Reported TOCTTOU Vulnerabilities

Domain	Application Name
Enterprise applications	Apache, bzip2, gzip, getmail, Imp-webmail, procmail, openldap, openSSL, Kerberos, OpenOffice, StarOffice, CUPS, SAP, samba
Administrative tools	at, diskcheck, GNU fileutils, log-watch, patchadd
Device managers	Esound, glint, pppd, Xinetd
Development tools	make, perl, Rational ClearCase, KDE, BitKeeper, Cscope

At the same time, TOCTTOU vulnerabilities are also a very challenging research problem due to their non-deterministic nature. They are very hard to detect because the occurrence of a TOCTTOU vulnerability requires a pair of certain system calls along the execution path of an application combined with appropriate environmental conditions. So they are more elusive than say, a buffer overflow bug which is only a single point of failure. TOCTTOU vulnerabilities are also hard to exploit, because they are essentially race condition errors so whether an attack can succeed relies on whether the attacking code is executed within the usually narrow window of vulnerability (on the order of

milliseconds as shown in section 4.2). Furthermore, normal static program analysis tools for detecting race conditions cannot be applied directly, since the attack programs are usually unavailable until the vulnerabilities are discovered.

The first contribution of this paper is a model-based approach to detecting TOCTTOU attacks in Unix-style operating systems. During the 10 years since the first systematic study of TOCTTOU problem by Bishop [2][3], only partial solutions have been proposed for some instances of the problem [5][6][13]. In this paper, we develop a model and list a comprehensive enumeration of TOCTTOU vulnerabilities for the Linux virtual file system. To the best of our knowledge, this is the most complete study of TOCTTOU problem so far.

The second contribution of the paper is a systematic search for potential TOCTTOU vulnerabilities in Linux system utility programs. We implemented model-based software tools that are able to detect previously reported TOCTTOU vulnerabilities as well as finding some unknown ones (e.g., in the *rpm* software distribution program, the *vi/vim* and *emacs* editors). We conducted a detailed experimental study of successfully exploiting these vulnerabilities and analyze the significant events during a TOCTTOU attack against the native binaries of *rpm* and *vi*. By repeating the experiments, we also evaluated the probability of these events happening, as well as the success rate of these non-deterministic TOCTTOU attacks. These analyses provide a quantitatively better understanding of TOCTTOU attacks.

The rest of the paper is organized as follows. Section 2 summarizes the CUU model of TOCTTOU vulnerabilities. Section 3 describes a framework that detects TOCTTOU vulnerabilities through monitoring of TOCTTOU pairs. Section 4 presents a detailed analysis of events during the attacks on *rpm* and *vi*, including a study of attack success probability. Section 5 discusses the accuracy of the detection software tools and shows the measured overhead incurred by the tools. Section 6 summarizes related work and Section 7 concludes the paper.

2 The CUU Model of TOCTTOU

2.1 Broad Definition of TOCTTOU

A necessary condition for a TOCTTOU vulnerability to happen is a pair of system calls (referred to as “TOCTTOU pair” in this paper) operating on the same disk object using a file pathname. The first system call (referred to as “CU-call”) establishes some preconditions about the file (e.g., the file exists, the current user has write privilege to the file, etc). The second system call (referred to as “Use-call”) operates on the file, based on those preconditions. In our model, the pre-

conditions about the file can be established either explicitly (e.g., **access** or **stat**) or implicitly (e.g., **open** or **creat**). Therefore, the TOCTTOU name is more restrictive than our model. Our model includes the original check-use system call pairs [2][3], plus use-use pairs. For example, a program may attempt to delete a file (instead of checking whether a file exists) before creating it. Consequently, the pair **<delete, create>** is also considered a (broadly defined) TOCTTOU pair.

2.2 An Enumeration of TOCTTOU pairs in Linux

We apply this model (called CUU) to the concrete situation of analyzing TOCTTOU problems in Linux. To get a complete list of TOCTTOU pairs, we first find the complete CUSet (the set of CU-calls) and UseSet (the set of Use-Calls). We select these two sets of kernel calls from the functional specification of Linux file system. We started from file system calls that require a pathname as input, and then filtered out those that are unlikely to be leveraged in a TOCTTOU attack. For example, **swapon** does not follow symbolic links so it is not included in the UseSet (Here we assume that all TOCTTOU attacks based on **swapon** are symbolic link kind attack). Finally we got the following CUSet and UseSet:

- CUSet = { **access, stat, open, creat, mknod, link, symlink, mkdir, unlink, rmdir, rename, execve, chmod, chown, truncate, utime, chdir, chroot, pivot_root, mount** }
- UseSet = { **creat, mknod, mkdir, rename, link, symlink, open, execve, chdir, chroot, pivot_root, mount, chmod, chown, truncate, utime** }

Although some system calls may appear unlikely candidates, they have been included after careful analysis. For example, **mknod** is in UseSet because it is able to create a new regular file, a function that is rarely known.

This classification of CUSet and UseSet is not structured enough for a complete analysis because some CU-calls and Use-calls are semantically unrelated. For example, **<creat, chdir>** is not a meaningful pair because **creat** creates a regular file while **chdir** expects a directory as argument. So we need to subdivide CUSet and UseSet so that a TOCTTOU pair at least applies to the same kind of storage objects (e.g. regular file, directory, or link). Thus we define the following sets.

Definition 1: CreationSet contains system calls that create new objects in the file system. It can be further divided into three subsets depending on the kind of objects that the system call creates:

CreationSet = FileCreationSet \cup LinkCreationSet \cup DirCreationSet, where

FileCreationSet = {creat, open, mknod, rename}
LinkCreationSet = {link, symlink, rename}
DirCreationSet = {mkdir, rename}

Definition 2: RemoveSet contains system calls that remove objects from the file system. It can be further divided into three corresponding subsets:

RemoveSet = FileRemoveSet \cup LinkRemoveSet \cup DirRemoveSet, where
FileRemoveSet = {unlink, rename}
LinkRemoveSet = {unlink, rename}
DirRemoveSet = {rmdir, rename}

Definition 3: NormalUseSet contains system calls which work on existing storage objects and do not remove them. We subdivide them into two sets:

NormalUseSet = FileNormalUseSet \cup DirNormalUseSet, where
FileNormalUseSet = {chmod, chown, truncate, utime, open, execve}
DirNormalUseSet = {chmod, chown, utime, mount, chdir, chroot, pivot_root}

Definition 4: CheckSet contains the system calls that establish preconditions about a file pathname explicitly.

CheckSet = {stat, access}

Using the above definitions, we divide the CUSet and UseSet into subsets:

CUSet = CheckSet \cup CreationSet \cup RemoveSet \cup NormalUseSet
UseSet = CreationSet \cup NormalUseSet

Based on the precondition established by the CU-call, we can divide the TOCTTOU pairs into two groups: Group 1 creates a new object and Group 2 operates on an existing object. We say that TOCTTOU vulnerabilities are *not* due to bad programming practices, since in Group 1 the CU-call establishes the precondition that the file pathname does not exist and in Group 2 the CU-call establishes the precondition that the file pathname exists.

Group 1 preconditions can be established either explicitly by CU-calls in the CheckSet, or implicitly by CU-calls in the RemoveSet. These are followed by Use-calls in a CreationSet of the corresponding type, e.g., the creation of a directory is only paired with a system call on a directory.

Group 1 = (CheckSet \times CreationSet) \cup (FileRemoveSet \times FileCreationSet) \cup (LinkRemoveSet \times LinkCreationSet) \cup (DirRemoveSet \times DirCreationSet).

Group 2 preconditions can be established by CU-calls in the CheckSet, or by CU-calls in the CreationSet (a file/directory/link exists after it is created), or by CU-calls in the NormalUseSet. These are followed by corresponding Use-calls. The link-related calls are paired with both FileNormalUseSet and DirNormalUseSet because a link can point to either a regular file or a directory.

Group 2 = (CheckSet \times NormalUseSet) \cup (FileCreationSet \times FileNormalUseSet) \cup (DirCreationSet \times DirNormalUseSet) \cup (LinkCreationSet \times FileNormalUseSet) \cup (LinkCreationSet \times DirNormalUseSet) \cup (FileNormalUseSet \times FileNormalUseSet) \cup (DirNormalUseSet \times DirNormalUseSet).

Intuitively, Group 1 \cup Group 2 completes the set of TOCTTOU pairs. A formal proof of the completeness of CUU is out of the scope of this paper and is addressed in another paper [20].

In summary, Table 2 shows these TOCTTOU pairs along two dimensions: the use of a storage object and whether the check was an explicit check or an implicit check. A total of 224 pairs have been identified using this table.

Table 2: Classification of TOCTTOU Pairs

Use	Explicit check	Implicit check
Create a regular file	CheckSet \times FileCreationSet	FileRemoveSet \times FileCreationSet
Create a directory	CheckSet \times DirCreationSet	DirRemoveSet \times DirCreationSet
Create a link	CheckSet \times LinkCreationSet	LinkRemoveSet \times LinkCreationSet
Read/Write/Execute or Change the attribute of a regular file	CheckSet \times FileNormalUseSet	(FileCreationSet \times FileNormalUseSet) \cup (LinkCreationSet \times FileNormalUseSet) \cup (FileNormalUseSet \times FileNormalUseSet)
Access or change the attribute of a directory	CheckSet \times DirNormalUseSet	(DirCreationSet \times DirNormalUseSet) \cup (LinkCreationSet \times DirNormalUseSet) \cup (DirNormalUseSet \times DirNormalUseSet)

2.3 Known TOCTTOU Examples

We applied our model to known TOCTTOU vulnerabilities and show the results in Table 3.

Table 3: Real world applications known to have TOCTTOU vulnerability

Applications	TOCTTOU pair	Classification
BitKeeper, Cscope 15.5, CUPS, getmail 4.2.0, glint, Kerberos 4, openldap, OpenOffice 1.0.1, patchadd, procmail, samba, Xinetd	<stat, open>	CheckSet × File-CreationSet
Rational ClearCase, pppd	<stat, chmod>	CheckSet × FileNormalUseSet
logwatch 2.1.1	<stat, mkdir>	CheckSet × Dir-CreationSet
bzip2-1.0.1, gzip, SAP	<open, chmod>	FileCreationSet × FileNormalUseSet
Mac OS X 10.4 – launchd	<open, chown>	
Apache 1.3.26, make	<open, open>	
StarOffice 5.2	<mkdir, chmod>	DirCreationSet × DirNormalUseSet

3 Model-Based TOCTTOU Detection

3.1 Components of Practical Attacks

An actual TOCTTOU vulnerability consists of a victim program containing a TOCTTOU pair (described in Section 2) and an attacker program trying to take advantage of the potential race condition introduced by the TOCTTOU pair. The attacker program attempts to access or modify the file being manipulated by the victim through shared access during the vulnerability window between the CU-call and Use-call. For example, by adding a line to an unintentionally shared script file in the *rpm* attack (Section 4.2), the attacker can trick the victim into executing unintended code at a higher privilege level (root). In general, we say that a TOCTTOU attack is profitable if the victim is running at a higher level of privilege. In Unix-style OSs, this means the victim running as root and the attacker as normal user.

An important observation is that even though the victim is running at a higher level of privilege, the attacker must have sufficient privileges to operate on the shared file attributes, e.g., creation or deletion. This observation narrows the scope of potential TOCTTOU vulnerabilities. Table 4 shows a list of directories owned by root in Linux. Since normal users cannot change the attributes or content of files in these directories, these files are safe.

Table 4: Directories Immune to TOCTTOU

/bin	/root	/usr/dict	/var/db
/boot	/proc	/usr/kerberos	/var/empty
/dev	/sbin	/usr/libexec	/var/ftp
/etc	/usr/bin	/usr/sbin	/var/lock
/lib	/usr/etc	/usr/src	/var/log
/misc	/usr/include	/usr/X11R6	/var/lib
/mnt	/usr/lib	/var/cache	/var/run
/opt			

3.2 CUU Model-Based Detection Tools

Based on the CUU model, we designed a software framework and implemented software tools to detect actual TOCTTOU vulnerabilities in Linux. Figure 1 shows the four components of our detection framework, based on dynamic monitoring of system calls made by sensitive applications (e.g., those that execute with root privileges). The first component of our framework is a set of plug-in Sensor code in the kernel, placed in system calls listed in the CUSet and UseSet (Section 2.2). These Sensors record the system call name and its arguments, particularly file name (full path for unique identification purposes). For some system calls, other related arguments are also recorded to assist in later analysis, e.g., the *mode* value of *chmod(path, mode)*. Some environmental variables are also recorded, including process id, name of the application, user id, group id, effective user id, and effective group id. This information will be used in the analysis to determine if a TOCTTOU pair can be exploited. We do not use standard Linux trace facilities such as *strace* for two reasons: First, *strace* does not output full pathname for files referred to using relative pathnames; Second, *strace* does not give enough environmental information such as effective user id.

The Sensors component also carries out a preliminary filtering of their log. Specifically, they identify the system calls on files under the system directories listed in Table 4 and filter them out, since those files are immune to TOCTTOU attacks. After this filter, remaining potentially vulnerable system calls are recorded in a circular FIFO ring buffer by *printk*.

The second component of our framework is the Collector, which periodically empties the ring buffer (before it fills up). The current implementation of the Collector is a Linux daemon that transforms the log records into an XML format and writes the output to a log file for both online and offline analysis.

The third component of our framework is the Analyzer, which looks for TOCTTOU pairs (listed in Table 2) that refer to the same file pathname. For offline analysis, this correlation is currently done using XSLT (eXtensible Stylesheet Language Transformations)

templates. This analysis proceeds in several rounds as follows.

Round 1: First, the Analyzer sorts the log records by file name, grouping its operation records such as the names and locations (sequence numbers) of system calls.

Round 2: Second, system calls on each file are paired to facilitate the matching of TOCTTOU pairs.

Round 3: Third, system call pairs are compared to the list in Table 2. When a TOCTTOU pair is found, an XSLT template is generated to extract the corresponding log records from the original log file.

Round 4: Fourth, the log records related to TOCTTOU pairs found are extracted into a new file for further inspection.

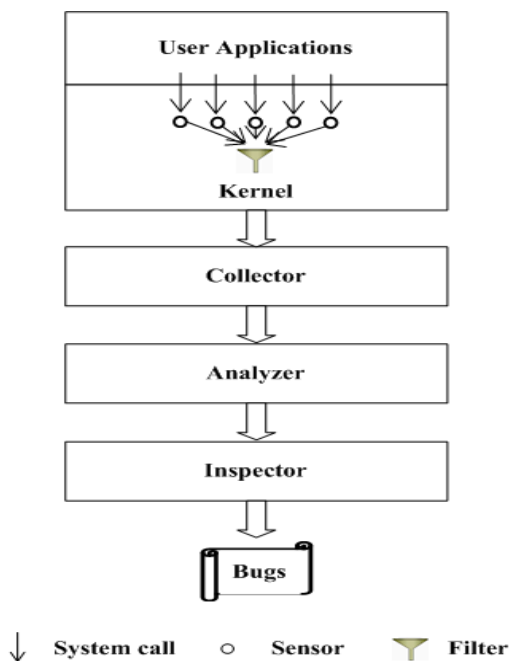


Figure 1: Framework for TOCTTOU Detection

The fourth component of our framework is the Inspector, which identifies the actual TOCTTOU vulnerability in the program being monitored. The Inspector links the TOCTTOU pair with associated environmental information, including file pathname, related arguments, process id, program name, user id, group id, effective user id, and effective group id. The Inspector decides whether an actual exploitation can occur.

For each TOCTTOU pair, the Inspector does the following steps:

- Check the arguments of the calls to see if these calls can be profitable to an attacker. For example,

if the Use-Call is **chmod**, then a value of 0666 for the *mode* argument falls into this category because this **chmod** can be used to make */etc/passwd* world-writable. On the other hand, a *mode* value of 0600 is not profitable because it will not give the attacker any permission on a file that he/she does not own. In this case the TOCTTOU pair in question is not a TOCTTOU vulnerability.

- Check the file pathname. For the **chmod** example, if the file is stored under a directory writable by an ordinary user, like his/her home directory, then continue to the next step; otherwise the TOCTTOU pair is not a TOCTTOU vulnerability.
- Check the effective user id. Continuing with the **chmod** example, if the effective user id is 0 (root), then report this TOCTTOU pair as a vulnerability; otherwise, the TOCTTOU pair is not a vulnerability.

It should be noted that the steps described above give only an outline of the Inspection process based on one attack scenario for one particular TOCTTOU pair. For different TOCTTOU pair and different attack scenario, the details of these checks can be different. For example, the same TOCTTOU pair as the above with a *mode* value of 0644 and the same other conditions is also considered a vulnerability because it can be exploited to make */etc/shadow* readable by an attacker. Thus the Inspector requires a template (or signature) for each kind of attack scenario. Table 5 shows the set of templates used by the current implementation of the Inspector. For brevity, this table does not show the file pathname and effective user id which are checked in every template. This set may be expanded as new attack scenarios are found.

Table 5: Templates used in the Inspector

Use-Call	Arguments to check	Sample attack scenarios
chmod	<i>mode</i>	Gain unauthorized access rights to <i>/etc/passwd</i>
chown	<i>owner, group</i>	Change the ownership of <i>/etc/passwd</i>
chroot		Access information under a restricted directory
execve		Run arbitrary code
open	<i>mode, flag</i>	Mislead privileged programs to do things for the attacker, or steal sensitive information
truncate	<i>length</i>	Erase the content of <i>/etc/passwd</i>

4 Analysis of Real TOCTTOU Attacks

4.1 Experimental Setup

We applied our detection framework and tools to find previously unreported TOCTTOU vulnerabilities in Linux. Although the CUU model describes all the TOCTTOU pairs in Linux file systems, it is impractical to test all the execution paths of all the system software (or even a single program of any complexity). Our intent is to learn as much as possible about real TOCTTOU vulnerabilities through a detailed analysis. The experiments show that significant weaknesses can be found relatively easily using our framework and tools.

From the discussion in Section 3.1, we focus our attention on system software programs that use file system (outside the directories listed in Table 4) as a root. Each program chosen is downloaded, installed, configured, and deployed. Furthermore, we also build a testing environment which includes the design and generation of a representative workload for each application, plus the analysis of TOCTTOU pairs observed. Although this is a laborious process that requires high expertise, one could imagine incorporating such testing environments into the software release of system programs, facilitating future evaluations and experiments.

Our tools were implemented on Red Hat 9 Linux (kernel 2.4.20) to find TOCTTOU vulnerabilities in about 130 commonly used utility programs. The script-based experiments consist of about 400 lines of shell script for 70 programs in /bin and /sbin. This script takes about 270 seconds to gather approximately 310K bytes of system call and event information. The other 60 programs were run manually using an interactive interface. From this sample of Linux system utilities, we found 5 potential TOCTTOU vulnerabilities (see Table 6).

The experiments were run on an Intel P4 (2.26GHz) laptop with 256M memory. The Collector produces an event log at the rate of 650 bytes/sec when the system is idle (only background tasks such as daemons are running), 11KB/sec during the peak time a large application such as OpenOffice is started, and 2KB/sec on average. The Analyzer processes the log at the speed of 4KB/sec.

From the list in Table 6, we wrote simple attack programs that confirmed the TOCTTOU vulnerabilities in *rpm*, *emacs* and *vi*. We discuss the attack on *rpm* and *vi* in detail (Sections 4.2 and 4.3, respectively), and outline the others in Section 4.4.

4.2 rpm 4.2 Temp File Vulnerability

rpm is a popular software management tool for install-

Table 6: Potential TOCTTOU Vulnerabilities

Application	TOCTTOU errors	Possible exploit
<i>vi</i>	<open, chown>	Changing the owner of /etc/passwd to an ordinary user
<i>rpm</i>	<open, open>	Running arbitrary command
<i>emacs</i>	<open, chmod>	Making /etc/shadow readable by an ordinary user
<i>gedit</i>	<rename, chown>	Changing the owner of /etc/passwd to an ordinary user
<i>esd</i> (Enlightened Sound Daemon)	<mkdir, chmod>	Gaining full access to another user's home directory

ing, uninstalling, verifying, querying, and updating software packages in Linux. When *rpm* installs or removes a software package, it creates a temporary script file in directories such as /var/tmp or /var/local/tmp. This shell script is used to install or remove help documentation of the software package. Since the access mode of this file is set to 666 (world-writable), an attacker can insert arbitrary commands into this script. Given the privileges required for installing software (usually root), this is a significant vulnerability. The TOCTTOU pair involved is <open, open>: the first **open** creates the script file for writing the script; and the second **open** is called in a child process to read and execute the script.

Table 7: Baseline vulnerability of rpm

Package Operation	Install (rpm -i)		Uninstall (rpm -e)	
	Average	Stdev	Average	Stdev
<i>t</i> (μsec)	125,188	9,930	110,571	10,961
<i>v</i> (μsec)	5,053	20	4,218	102
<i>v/t</i>	4.1%	---	3.8%	---

4.2.1 Baseline Analysis of rpm

In our evaluation of the TOCTTOU vulnerability in *rpm*, we start by measuring the total running time of *rpm* (denoted by *t*) and the window of vulnerability (the time interval between the two **opens**, denoted by *v*). We ran *rpm* (as root) 100 times, alternatively installing and uninstalling a package named *sharutils-4.2.1-14.i386.rpm*, and measured *t* and *v* for each invocation. From Table 7 we can see that the window of vulnerability is relatively narrow (less than 5%), since the two **opens** are separated only by a few milliseconds.

4.2.2 An Experiment to Exploit *rpm*

The second part of our evaluation is to measure the effectiveness of an attack trying to exploit this apparently small window of vulnerability. This experiment runs a user-level attack process in a loop. It constantly checks for the existence of a file name with the prefix “/var/tmp/rpm-tmp”. A victim process (*rpm* run by root) installs a software package and creates a script file of that name. Note that *rpm* inserts a random suffix as protection against direct guessing, but a directory listing command bypasses the need to guess the full pathname. If a file name of the expected prefix appears, the attacker appends the command “*chown* attacker:attacker /etc/passwd” to it. If the append happens during the window of vulnerability, then the child process of *rpm* will execute the script and the inserted command line, making the attacker the owner of /etc/passwd. When *rpm* finishes, the test program checks whether the attacker has become the owner of /etc/passwd.

Due to the non-deterministic nature of these experiments, we ran the experiment 100 times in a batch. After running several batches, we found a surprisingly high average number of 85 successful attacks per batch, considering the apparently narrow window of vulnerability shown in Table 7.

4.2.3 Event Analysis of *rpm* Exploit

To fully understand what happened during the TOCTTOU attack, we analyze the important system events during the experiment. Figure 2 shows the events in a successful exploit of *rpm*. In Figure 2, the dark (upper) line shows the events of the *rpm* process, and the lower line shows the events of the attacker process. The attacker process stays in a loop looking for file names of interest. When the *rpm* process creates the file (just before the 200 msec clock tick), the attacker detects it and appends the *chown* line to the temporary script and goes back to the loop.

The two timelines show that even though the CPU consumption during the window of vulnerability is relatively small, the *rpm* process causes interrupts that lengthen the window, represented by dotted upper line. Specifically, there are at least two scheduling actions within the *rpm* vulnerability window: *rpm* creates a new process to execute *bash*, which creates another new process to execute an external executable file (/sbin/install-info). Each process creation causes *rpm* to yield CPU to the scheduler. Figure 2 shows that the attacker process is scheduled as a result and the attack succeeds. Consequently, the two scheduling actions created by *rpm* make the attack more likely to succeed because *rpm* yields the CPU in the window of vulnerability.

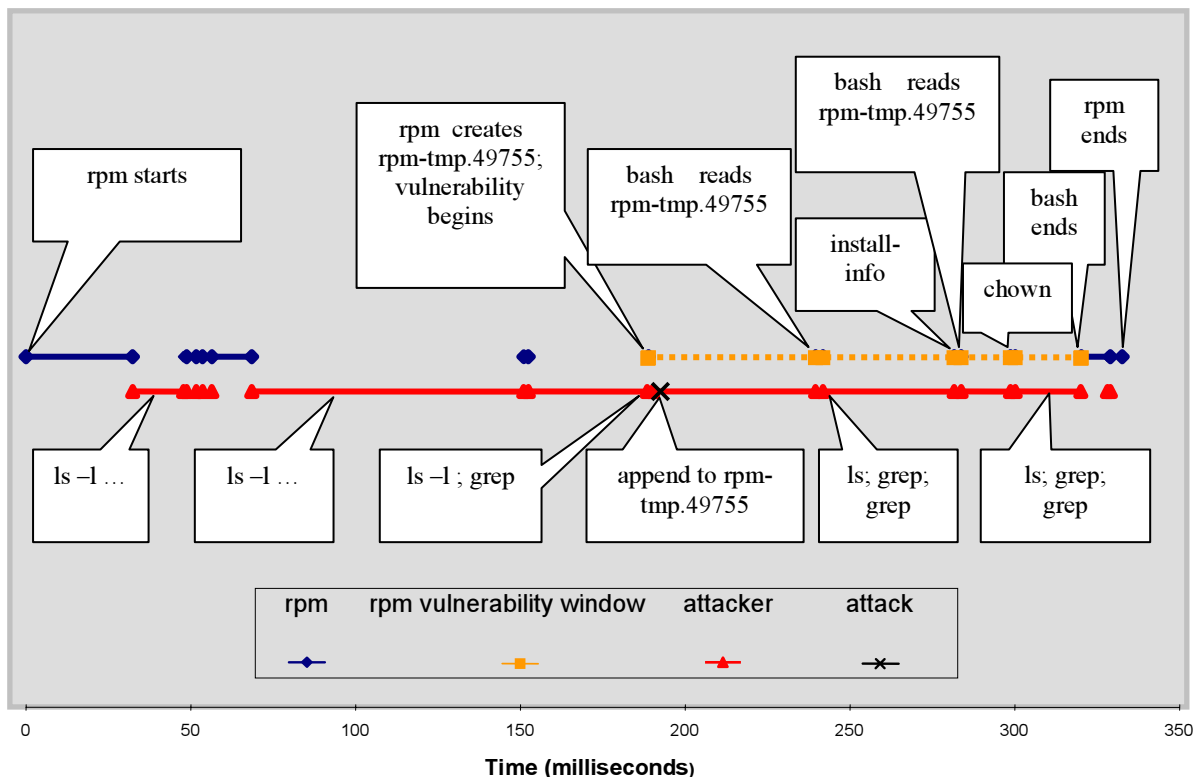


Figure 2: Event Analysis of *rpm* Exploit

In our experiments, we also found another reason more attacks succeed than indicated by the short window of vulnerability. Specifically, we observed that in some cases the appending to the script file by the attacker happened after the second **open** of *rpm* (outside the window), but the attack still succeeds. In these cases, we believe that append started after *bash* opened the script file (the second **open** of *rpm*), but it finished before *bash* reached the end of the script. Since *bash* interprets the script line by line, there is a good chance of executing the newly appended line. These two explanations (CPU yielding and slow interpretation of the script) help explain the lengthening of the window of vulnerability and the high attack success rate of 85%.

4.3 *vi* 6.1 Vulnerability

The Unix “visual editor” *vi* is a widely used text editor in many UNIX-style environments. For example, Red Hat Linux distribution includes *vi* 6.1. Using our tools, we found potential TOCTTOU vulnerabilities in *vi* 6.1. Specifically, if *vi* is run by root to edit a file owned by a normal user, then the normal user may become the owner of sensitive files such as */etc/passwd*.

The problem can be summarized as follows. When *vi* saves the file being edited, it first renames the original file as a backup, then creates a new file with the original name. The new file is closed after all the content in the edit buffer is written. If *vi* is running as root, the initial owner and group of this new file is root, so *vi* needs to change the owner and group of the new file to its original owner and group. This forms an **<open, chown>** window of vulnerability every time *vi* saves the file. During this window, if the file name can be changed to a link to */etc/passwd*, then *vi* can be tricked into changing the ownership of */etc/passwd* to the normal user.

4.3.1 Baseline Analysis of *vi*

Using the same method of the *rpm* study, we measured the percentage of time when *vi* is running in its vulnerability window as it saves the file being edited. In *vi*, this depends on the edited file size. In our experiments, we bypass the user typing time to avoid the variations caused by human participation.

We define the save window *t* as the time *vi* spends in processing one “save” command, and the vulnerability window *v* during which TOCTTOU attack may happen. We measured 60 consecutive “saves” of the file for *t*, and timestamp the **open** and **chown** system calls for *v*. Since the “save” time of a file depends on the file size, we did a set of experiments on different file sizes. Figure 4 shows the time required for a “save” command for files of sizes from 100KB to 10MB. We found a per file fixed cost that takes about 14msec for the small (100KB) file and an incremental cost of 9msec/MB (for files of size up to 10MB).

Since **chown** happens after the file is completed, the window of vulnerability *v* follows approximately the same incremental growth of 9msec/MB (see Figure 4). Figure 3 shows the window of vulnerability to be relatively long compared to the total “save” time. It gradually grows to about 80% of the “save” total elapsed time for 10MB files. This experiment tells us that *vi* is more vulnerable when the file being edited is larger. For a small file (100KB size) the window of vulnerability is still about 5% of the “save” time.

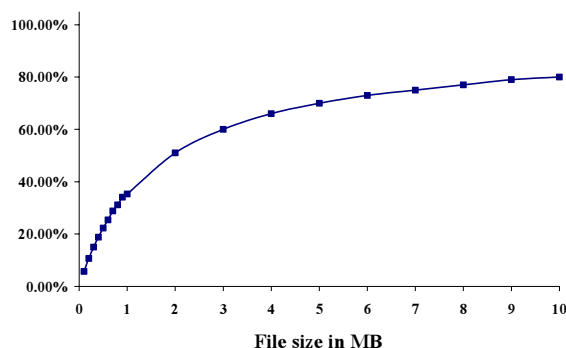


Figure 3: Window of Vulnerability Divided by Total Save Time, as a Function of File Size

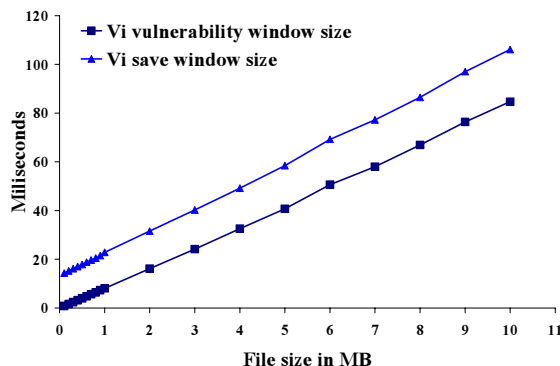


Figure 4: Vulnerability and Save Window Sizes of *vi*

4.3.2 An Experiment to Exploit *vi*

Unlike a batch program such as *rpm*, which is easily run from a script, *vi* is designed for interactive use by humans. To eliminate the influence of human “think time” in the experiments, we wrote another program to interact with *vi* by sending it commands that simulate human typing. This reduces the run-time and the window of vulnerability to minimum. The experiment runs a *vi* (as root) editing a file owned by the attacker in the attacker’s home directory. The editing consists of either appending or deleting a line from the file and the experiment ends with *vi* exiting.

The attack consists of a tight loop constantly checking (by `stat`-ing) whether the owner of the file has become root, which signifies the start of the window of vulnerability. Once this happens, the attacker replaces the file with a symbolic link to `/etc/passwd` (as shown in Figure 7). When `vi` exits, it should change the ownership of `/etc/passwd` to the attacker. The attacker program checks for this ownership change. If `vi` finishes and `/etc/passwd` is still owned by root, the attack fails.

Contrary to the surprisingly high probability of success in the `rpm` case, we found a relatively low probability of success in the `vi` case (see Figure 5 and Figure 6), despite a relatively wide window of vulnerability. This leads to a more careful analysis of the system events during the attack.

4.3.3 Event Analysis of `vi` Exploit

Although the window of vulnerability may be wide, an attack will succeed only when:

1. `vi` has called `open` to create the new file,
2. `vi` has not called `chown`,
3. `vi` relinquishes CPU, voluntarily or involuntarily, and the attacker is scheduled to run, and
4. the attacker process finishes the file redirection during this run.

The first two conditions have been studied in the baseline experiment. The fourth condition depends on the implementation of the attacker program. For example, if the attacker program is written in C instead of shell script, it will be less likely to be interrupted.

The third condition is the least predictable. In our experiments, we have found several reasons for `vi` to relinquish CPU. First, `vi` may suspend itself to wait for I/O. This is likely since the window of vulnerability includes the writing of the content of the file, which may result in disk operations. Second, `vi` may use up its CPU slice. Third, `vi` may be preempted by higher priority processes such as `ntpd`, `kswapd`, and `bdflush` kernel threads. Even after `vi` relinquishes CPU, the second part of the condition (that the attacker process is scheduled to run) still depends on other processes not being ready to run.

This analysis illustrates the highly non-deterministic nature of a TOCTTOU attack. To achieve a statistically meaningful evaluation, we repeat the experiments and compute the probability of attack success. To make the experimental results reproducible, we eliminated all the confounding factors that we have identified. For example, in each round of experiments, we ran `vi` at least 50 times, each time on a different file, to minimize file caching effects. We also observed memory allocation problems after large files

have been used. To relieve memory pressure, we added a 2-second delay between successive `vi` invocations.

Figure 5 shows the success rate for file sizes ranging from 100KB to 1MB averaged over 500 rounds. We see that for small files, there is a rough correlation between the size of window of vulnerability and success rate. Although not strictly linear, the larger the file being edited, the larger is the probability of successfully attacking `vi`.

Figure 6 shows the results for file sizes ranging from 2MB to 4MB, with a stepping size of 20KB, averaged over 100 rounds. Unlike the dominantly increasing success rate for small file sizes, we found apparently random fluctuations on success rates between file sizes of 2MB and 3MB, probably due to race conditions. For example, files of size 2MB have success rate of 4%, which is lower than the 8% success rate of file size 500KB in Figure 5. The growing success trend resumes after files become larger than 3MB.

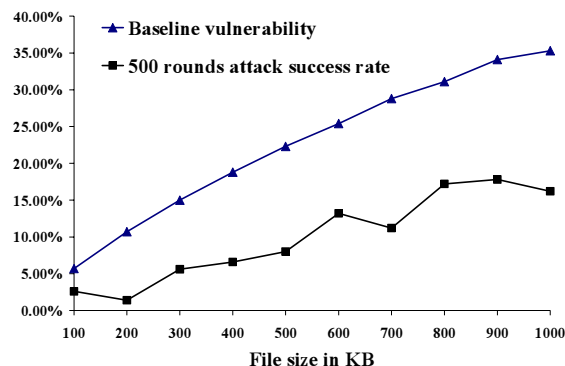


Figure 5: Success Rate of Attacking `vi` (small files)

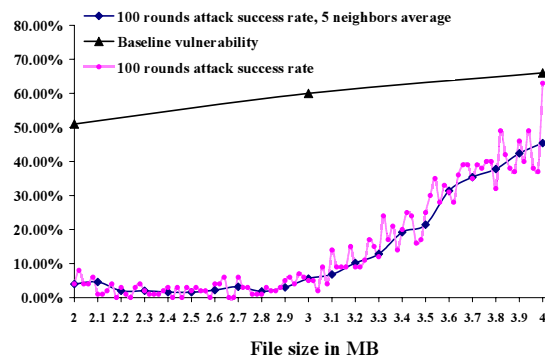


Figure 6: Success Rates of Attacking `vi` (large files)

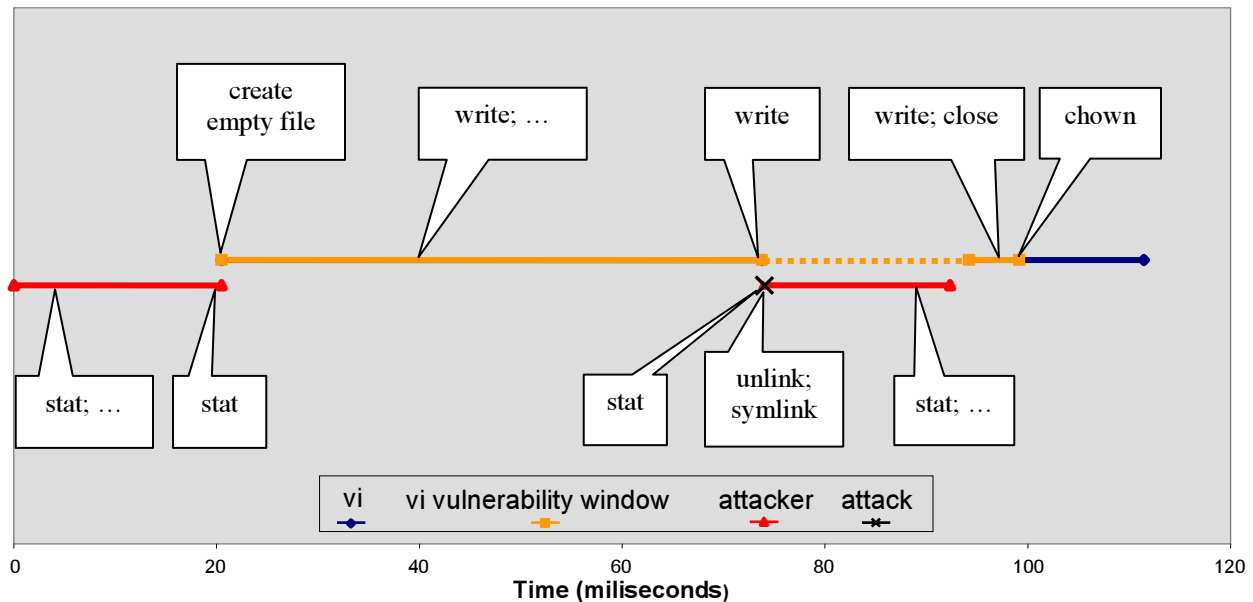


Figure 7: Event Analysis of the *vi* Exploit

4.4 Other Vulnerabilities

In our experiments, we identified 5 TOCTTOU pairs (see Table 6) and confirmed 3 of them through direct attacks (*rpm*, *vi*, and *emacs*). Due to its similarity to the *vi* experiments (Section 4.3), the analysis of the attack of *emacs* is omitted here.

We also tried to attack *gedit*, the fourth vulnerability discovered, but we found a very low probability of successful attack. Like *vi*, *gedit* becomes vulnerable when it saves the file being edited. Unlike *vi*, *gedit* writes to a temporary scratch file, then renames the scratch file to the original file name, and calls **chown**. Thus the window of vulnerability is between the **rename** and the directly following **chown**, a very short time that reduces the probability of successful attack. A full analysis of *gedit* experiments is beyond the scope of this paper.

The fifth vulnerability is the Enlightened Sound Daemon (*esd*), which creates a directory `/tmp/.esd` then changes the access mode of this directory to `777`, giving full permissions (read/write/execute) to all users. Besides, this directory is under `/tmp`, a place where any user can create files or directories. So a possible attack is to create a symbolic link `/tmp/.esd` before the **mkdir** call of *esd* and let the link point to some directories owned by the running user (such as his/her home directory). If *esd* does not check whether its **mkdir** call succeeds, then it will change the access mode of the running user's home directory to `777`. Then an attacker has full access to the running user's home directory. We postponed our experiments on *esd* since this

TOCTTOU vulnerability has been reported in BUGTRAQ [17].

Overall, we consider the CUU model-based detection framework to be a success. With a modest number of experiments, we confirmed known TOCTTOU vulnerabilities and found several previously unreported ones. However, this offline analysis only covers the execution paths exercised by the workloads, so it cannot guarantee the absence of TOCTTOU vulnerabilities when none is reported.

In this paper, our research focuses on the scheduling aspects of TOCTTOU attacks in uniprocessor environments. Multiprocessors, hyper-threaded uniprocessors, or multi-core processors are beyond the scope of this paper and subject of ongoing research.

5 Evaluation of Detection Method

5.1 Discussion of False Negatives

As mentioned in Section 4.1, our tools are not designed for exhaustive testing. While we attempted to generate representative workloads for the 130 programs tested, we cannot guarantee coverage of all execution paths. The coverage problem may be alleviated by improvements in the testing technology and documentation.

More fundamentally, the CUU-Model covers pairs of file system calls, assuming that a precondition is established by the CU-call before the Use-call relies on it. In programs where preconditions are not explicitly established (a bad programming practice), e.g., a program creates a temporary file under a known name without first **stat**-ing the existence of the file, exploits may

happen outside the CUU model. The problem of complex interactions among more than a pair of system calls is an open research question. (Currently, there are no known examples of such complex vulnerabilities.)

5.2 Discussion of False Positives

Tool-based detection of vulnerabilities typically does not achieve 100% precision. The framework described in Section 3 is no exception. There are some technical sources of false positives:

1. Incomplete knowledge of search space: The list of immune directories (Table 4) is not complete because of the dynamic changes to system state (e.g. newly created root-owned directories under `/usr/local`), which leads to false positives.
2. Artifacts of test environment: If the test cases themselves use `/tmp` or the home directory of an ordinary user, our tools have to report related TOCTTOU pairs, which are false positives. For example, the initial test case for *cpio* uses a temporary directory `/tmp/cpio`, so the tools reported a `<stat, chdir>` on this directory.
3. Coincidental events: Because our tools do system-wide monitoring, they capture file system calls made by every process. Sometimes two unrelated processes happen to make system calls on the same file that appear to be a TOCTTOU pair.
4. Incomplete knowledge of application domain: Not every TOCTTOU pair is profitably exploitable. For example, the application *rpm* invoked by “--addsign” option contains a `<stat, open>` pair, which can open any file in the system for reading, such as `/etc/shadow`. However, *rpm* can not process `/etc/shadow` because it is not in the format recognizable by *rpm*. So it is unlikely that this pair can be exploited to undermine a system.

By improving the kernel filter (source 1), re-designing test cases (source 2), and reducing concurrent activities (source 3), we reduced the false positive of our tools; for example, in one experiment testing 33 Linux programs under `/bin`, the false positive rate fell from 75% to 27%. However, source 4 is hard to remove due to the differences among application domains.

5.3 Overhead Measurements

To evaluate the overhead of our detection framework, we ran a variant of the Andrew benchmark [9]. The benchmark consists of five stages. First, it uses **mkdir** to recursively create 110 directories. Second it copies 744 files with a total size of 12MB. Third, it **stats** 1715 files and directories. Fourth, it *greps* (scan through) these files and directories, reading a total amount of 26M bytes. Fifth, it does a compilation of around 150 source files. For every stage, the total running time is calculated and recorded. We run this benchmark for 20 rounds and get the average. To mitigate the interference

from other processes during the run, we start Red Hat in single-user mode (without X window system and daemon processes such as *apmd*, *crond*, *cardmgr*, *syslogd*, *gpm*, *cups* and *sendmail*). To get an estimation of the overhead of our system, we run this experiment on a Linux box without modifications to get the baseline results, and then a Linux box with our monitoring tools (without the Analyzer and the Inspector which are used offline). For the latter case, we ran the experiment under two different directories to see the influence of file pathname to the overhead. The total running time of these five stages for the experiments is shown in Figure 8 and Table 8.

The results show a relatively higher overhead for **mkdir**, **copy** and **stat** when the benchmark is run under an ordinary user’s home directory (denoted Vulnerable Dir in Figure 8 and Table 8). But when the benchmark is run under `/root` (denoted Immune Dir in Figure 8 and Table 8), the overhead becomes much lower (dropping from 144% to 14% for **stat**). This difference shows that **printks** in the kernel and the Collector daemon process contribute significantly to the overhead, because the filter in kernel suppresses most log messages caused by the benchmark when it runs in a directory immune to TOCTTOU (Table 4), therefore the **printks** and Collector have much less work to do. The other source of overhead comes from the Sensor (including the filter and a query of the internal `/proc` file system data structure to map a process id to the complete command line to assist the Inspector). However, the overhead of our detection tools is amortized by application workload, as shown for compilation.

PostMark benchmark [11] is designed to create a large pool of continually changing files and to measure the transaction rates for a workload approximating a large Internet electronic mail server. Since mail server software such as *sendmail* had well known TOCTTOU problems, PostMark seems to be another representative workload to evaluate the performance overhead of our software tools.

When PostMark benchmark is running, it first tests the speed of creating new files, and the files have variable lengths that are configurable. Then it tests the speed of transactions. Each transaction has a pair of smaller transactions, which are either read/append or create/delete.

On the original Linux kernel the running time of this benchmark is 30 seconds. On our modified kernel, with all the same parameter settings, the running time is 30.35 seconds when the experiment is run under `/root` (an immune directory), and 35 seconds when the experiment is run under a vulnerable directory. So the overhead is 1.17% and 16.7% for these two cases, respectively. This result also shows that the **printks** and the Collector contribute significantly to the overhead.

Table 8: Andrew Benchmark Results (msec)

Functions	Original Linux	Modified Linux Immune Dir		Modified Linux Vulnerable Dir	
		Time	Overhead	Time	Overhead
mkdir	2.8 ± 0.06	3.0 ± 0.10	7.1%	4.1 ± 0.05	46%
copy	59.2 ± 0.49	64.8 ± 2.2	9.5%	80.8 ± 0.46	36%
stat	61.1 ± 0.55	69.4 ± 0.41	14%	149.3 ± 3.5	144%
grep	543.1 ± 2.4	576.2 ± 5.9	6.1%	645.3 ± 3.7	19%
compile	20,668 ± 66	20,959 ± 90	1.4%	21,311 ± 195	3.1%

6 Related Work

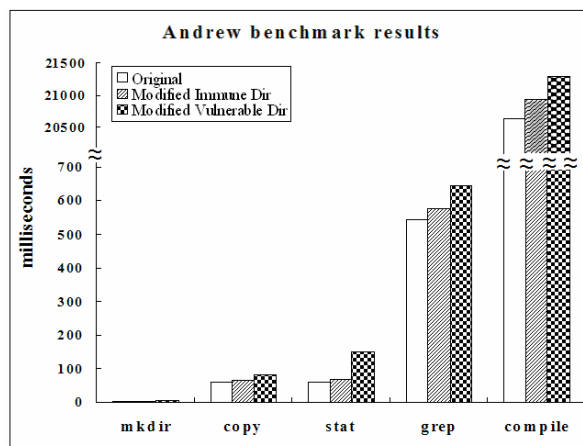
The impression that TOCTTOU vulnerabilities are due to bad programming practices is probably created by the patches and solutions suggested in advisories and reports on TOCTTOU exploits from US-CERT [14], CIAC [15] and BUGTRAQ [16]. For example, many of the reported problems link temporary files to another file to be manipulated. Examples of TOCTTOU pairs are **<stat, mkdir>** and **<stat, open>**. Typical solutions suggested for patching these problems include:

- Using random number to obfuscate file names.
- Replacing mktemp() with mkstemp().
- Using a strict umask to protect temporary directories.
- Dropping privileges to those of an explicitly configured user.
- Setting proper file/directory permissions.
- Checking the return code of function calls.

Although these suggestions are useful, they cannot detect nor prevent these exploits. CUU model provides a systematic approach to detection and prevention (outside the scope of this paper).

In recent years, static analysis of source code has been used to find bugs in systems software. Significant examples include: Bishop and Dilger's prototype analysis tool that used pattern matching to look for TOCTTOU pairs [2][3]; Meta-compilation [7] using compiler-extensions to check conformance to system specific rules; RacerX [8] decoupling compilation from rule-checking plus inter-procedural analysis; MOPS [4] using model checking to verify program security properties. These static analysis tools are limited in the detection of real TOCTTOU problems due to difficulties with dynamic states (e.g., file names, ownership, and access rights) and unavailability of attacker programs for race condition checking.

Dynamic monitoring and analysis have been used to gain insights into a system's behavior in many different

**Figure 8: Andrew Benchmark Results**

settings, such as file access prediction in mobile computing [18][19]. In the particular area of software security, dynamic monitors observe application execution to find software bugs. These tools can be further classified into dynamic online analysis tools and post mortem analysis tools. Eraser [12] is an online analysis tool that uses lockset analysis to find race conditions (unsynchronized access to shared variables) in a multi-threaded program. Calvin et al [10] proposed a post mortem analysis tool for security vulnerabilities (including TOCTTOU) related to privileged programs (setuid programs). However, this tool can only detect the result of exploiting a TOCTTOU vulnerability but cannot locate the error.

In the area of general mechanisms to defend against TOCTTOU attacks, solutions have been proposed for specific TOCTTOU pairs. For example, Dean and Hu [6] add multiple **<access, open>** pairs (called strengthening rounds) to reduce the probability of successful attack against the TOCTTOU pair **<access, open>**. RaceGuard [5] prevents the temporary file creation race condition in UNIX systems, specifically, the **<stat, open>** TOCTTOU pair. Tsyklevich and Yee [13] described a protection mechanism (called pseudo-transaction) to prevent race conditions between specified system call pairs. Although the pseudo-transaction mechanism is sufficiently general, their specification of TOCTTOU pairs was based on heuristics. The CUU model is a generalization of previous work watching for specific TOCTTOU pairs. Our work also complements mechanisms such as pseudo-transactions by providing a complete model (with 224 identified TOCTTOU pairs) to monitor all potentially dangerous interactions.

7 Conclusion

According to CERT [14] advisories and BUGTRAQ [16] reports, TOCTTOU problems are both numerous

and serious. We describe the CUU model and framework to detect TOCTTOU vulnerabilities. The model consists of 224 pairs of dangerous file system calls (the TOCTTOU pairs) and we implemented the detection framework for offline analysis of TOCTTOU vulnerabilities. The CUU model is programming language-independent. The software tools work without changes or access to application source code.

Using offline analysis, we confirmed known TOCTTOU attacks such as *esd* [17]. Running a relatively modest set of experiments (about 130 system utility programs), we also found and confirmed previously unreported TOCTTOU vulnerabilities in (the unmodified, original version of) *rpm*, *emacs* and *vi*.

To understand better TOCTTOU vulnerabilities, we recorded and analyzed in detail the main events in the attack scenarios. These analyses support a quantitative evaluation of the likelihood of success for each attack (ranging from very unlikely in the *gedit* case to highly likely in the *rpm* case at 85%). This evaluation is a non-trivial task for non-deterministic concurrent programs. We also measured and found modest performance overhead of our tools by running the Andrew and PostMark benchmarks (a few percent additional overhead for application level benchmarks).

The CUU model-based analysis of TOCTTOU vulnerabilities also suggests online defense mechanisms similar to pseudo-transactions [13]. This is a topic of active research and beyond the scope of this paper.

8 Acknowledgement

This work was partially supported by NSF/CISE IIS and CNS divisions through grants CCR-0121643, IDM-0242397 and ITR-0219902. We also thank the anonymous FAST reviewers for their insightful comments.

9 References

- [1] R. P. Abbott, J.S. Chin, J.E. Donnelley, W.L. Konigsford, S. Tokubo, and D.A. Webb. Security Analysis and Enhancements of Computer Operating Systems. NBSIR 76-1041, Institute of Computer Sciences and Technology, National Bureau of Standards, April 1976.
- [2] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996.
- [3] Matt Bishop. Race Conditions, Files, and Security Flaws; or the Tortoise and the Hare Redux. Technical Report 95-8, Department of Computer Science, University of California at Davis, September 1995.
- [4] Hao Chen, David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, Washington, DC, November 2002.
- [5] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel Protection From Temporary File Race Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington DC, August 2001.
- [6] Drew Dean and Alan J. Hu. Fixing Races for Fun and Profit: How to use access(2). In *Proceedings of the 13th USENIX Security Symposium*, San Diego, CA, August 2004.
- [7] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallett. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of Operating Systems Design and Implementation (OSDI)*, September 2000.
- [8] Dawson Engler, Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP'2003)*.
- [9] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system, *Transactions on Computer Systems*, vol. 6, pp. 51–81, February 1988.
- [10] Calvin Ko, George Fink, Karl Levitt. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring. *Proceedings of the 10th Annual Computer Security Applications Conference*, page 134–144.
- [11] PostMark benchmark. http://www.netapp.com/tech_library/3022.html
- [12] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, Vol. 15, No. 4, November 1997, Pages 391–411.
- [13] Eugene Tsyrlkevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, pages 243–256, Washington, DC, August 2003.
- [14] United States Computer Emergency Readiness Team, <http://www.kb.cert.org/vuls/>
- [15] U.S. Department of Energy Computer Incident Advisory Capability. <http://www.ciac.org/ciac/>
- [16] BUGTRAQ Archive
<http://msgs.securepoint.com/bugtraq/>
- [17] BUGTRAQ report RHSA-2000:077-03: esound contains a race condition. <http://msgs.securepoint.com/bugtraq/>
- [18] Ahmed Amer and Darrell D. E. Long. Noah: Low-cost file access prediction through pairs. In *Proceedings of the 20th IEEE International Performance, Computing and Communications Conference (IPCCC '01)*, April 2001.
- [19] Tsozen Yeh, Darrell D. E. Long, and Scott Brandt. Performing file prediction with a program-based successor model. In *Proceedings of the 9th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*, pages 193–202, Cincinnati, OH, August 2001.
- [20] Calton Pu, Jinpeng Wei. A theoretical study of TOCTTOU problem modeling. In preparation.

A Security Model for Full-Text File System Search in Multi-User Environments

Stefan Büttcher and Charles L. A. Clarke

*School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada*

Abstract

Most desktop search systems maintain per-user indices to keep track of file contents. In a multi-user environment, this is not a viable solution, because the same file has to be indexed many times, once for every user that may access the file, causing both space and performance problems. Having a single system-wide index for all users, on the other hand, allows for efficient indexing but requires special security mechanisms to guarantee that the search results do not violate any file permissions.

We present a security model for full-text file system search, based on the UNIX security model, and discuss two possible implementations of the model. We show that the first implementation, based on a postprocessing approach, allows an arbitrary user to obtain information about the content of files for which he does not have read permission. The second implementation does not share this problem. We give an experimental performance evaluation for both implementations and point out query optimization opportunities for the second one.

1 Introduction and Overview

With the advent of desktop and file system search tools by Google, Microsoft, Apple, and others, efficient file system search is becoming an integral component of future operating systems. These search systems are able to deliver the response to a search query within a fraction of a second because they index the file system ahead of time and keep an index that, for every term that appears in the file system, contains a list of all files in which the term occurs and the exact positions within those files (called the term's *posting list*).

While indexing the file system has the obvious advantage that queries can be answered much faster from the index than by an exhaustive disk scan, it also has the obvious disadvantage that a full-text index requires significant disk space, sometimes more than what is avail-

able. Therefore, it is important to keep the disk space consumption of the indexing system as low as possible. In particular, for a computer system with many users, it is infeasible to have an individual index for every user in the system. In a typical UNIX environment, for example, it is not unusual that about half of the file system is readable by all users in the system. In such a situation, even a single `chmod` operation – making a previously private file readable by everybody – would trigger a large number of index update operations if per-user indices were used. Similarly, due to the lack of information sharing among the individual per-user indices, multiple copies of the index information about the same file would need to be stored on disk, leading to a disk space consumption that could easily exceed that of the original file.

We investigated different desktop search tools, by Google¹, Microsoft², Apple³, Yahoo⁴, and Copernic⁵, and found that all but Apple's Spotlight maintain a separate index for every user (Google's search tool uses a system-wide index, but this index may only be accessed by users with administrator rights, which makes the software unusable in multi-user environments). While this is an unsatisfactory solution because of the increased disk space consumption, it is very secure because all file access permissions are automatically respected. Since the indexing process has the same privileges as the user that it belongs to, security restrictions cannot be violated, and the index accurately resembles the user's view of the file system.

If a single system-wide index is used instead, this index contains information about all files in the file system. Thus, whenever the search system processes a search query, care has to be taken that the results are consistent with the user's view of the file system. A search result is obviously inconsistent with the user's view of the file system if it contains files for which the user does not have read permission. However, there are more subtle cases of inconsistency. In general, we say that the result to a search query is inconsistent with the user's view of the

file system if some aspect of it (e.g., the order in which matching files are returned) depends on the content of files that cannot be read by the user. Examples of such inconsistencies are discussed in section 5.

An obvious way to address the consistency problem is the postprocessing approach: The same, system-wide index is used for all users, and every query is processed in the same way, regardless of which user submitted the query; after the query processor has computed the list of files matching the query, all file permissions are checked, and files that may not be searched by the user are removed from the final result. This approach, which is used by Apple's Spotlight search system (see the Apple Spotlight technology brief⁶ for details), works well for Boolean queries. However, pure Boolean queries are not always appropriate. If the number of files in a file system is large, the search system has to do some sort of relevance ranking in order to present the most likely relevant files first and help the user find the information he is looking for faster. Usually, a TF/IDF-based (term frequency / inverse document frequency) algorithm is used to perform this relevance ranking.

In this paper, we present a full-text search security model. We show that, if a TF/IDF-style ranking algorithm is used by the search system, an implementation of the security model must not follow the postprocessing approach. If it does, it produces search results that are inconsistent with the user's view of the file system. The inconsistencies can be exploited by the user in a systematic way and allow him to obtain information about the content of files which he is not allowed to search. While we do not know the exact ranking algorithm employed by Apple's Spotlight, we conjecture that it is at least in parts based on the TF/IDF paradigm (as TF/IDF-based algorithms are the most popular ranking techniques in information retrieval systems) and therefore amenable to the attacks described in this paper.

After discussing possible attacks on the postprocessing approach, we present a second approach to the inconsistency problem which guarantees that all search results are consistent with the user's view of the file system and which therefore does not allow a user to infer anything about the content of files which he may not search. This safe implementation of the file system search security model is part of the Wumpus⁷ file system search engine. The system is freely available under the terms of the GNU General Public License.

In the next two sections, we give a brief overview of previous work on security issues in multi-user environments (section 2) and an introduction to basic information retrieval techniques (section 3). This introduction covers the Okapi BM25 relevance ranking function (section 3.2) and the structural query language GCL (section 3.3) on which our retrieval framework and the safe im-

plementation of the security model are based.

In section 4, we present a general file system search security model and define what it means for a file to be *searchable* by a user. Section 5 discusses the first implementation of the security model, based on the postprocessing approach described above. We show how this implementation can be exploited in order to obtain the total number of files in the file system containing a certain term. This is done by systematically creating and deleting files, submitting search queries to the search system, and looking at either the relevance scores or the relative ranks of the files returned by the search engine.

In section 6, we present a second implementation of the security model. This implementation is immune against the attacks described in section 5. Its performance is evaluated experimentally in section 7 and compared to the performance of the postprocessing approach. Opportunities for query optimization are discussed in section 8, where we show that making an almost non-restrictive assumption about the independence of different files allows us to virtually nullify the overhead of the security mechanisms in the search system.

2 Related Work

While some research has been done in the area of high-performance dynamic indexing [BCC94] [LZW04], which is also very important for file system search, the security problems associated with full-text search in a multi-user environment have not yet been studied.

In his report on the major decisions in the design of Microsoft's Tripoli search engine, Peltonen [Pel97] demands that "full text indexing must never compromise operating or file system security". However, after this initial claim, the topic is not mentioned again in his paper. Turtle and Flood [TF95] touch the topic of text retrieval in multi-user environments, but only mention the special memory requirements, not the security requirements.

Griffiths and Wade [GW76] and Fagin [Fag78] were among the first who investigated security mechanisms and access control in relational database systems (System R). Both papers study discretionary access control with ownership-based administration, in some sense similar to the UNIX file system security model [RT74] [Rit78]. However, their work goes far beyond UNIX in some aspects. For example, in their model it is possible that a user grants the right to grant rights for file (table) access to other users, which is impossible in UNIX. Bertino et al. [BJS95] give an overview of database security models and access control mechanisms, such as group authorization [WL81] and authorization revocation [BSJ97].

While our work is closely related to existing research in database security, most results are not applicable to the file system search scenario because the requirements of a relational database are different from those of a file search system and because the security model of the search system is rather strictly predetermined by the security model of the underlying file system, UNIX in our case, which does not allow most of the operations database management systems support. Furthermore, the optimizations discussed in section 8 cannot be realized in a relational database systems.

3 Information Retrieval Basics

In this section, we give an introduction to basic information retrieval techniques. We start with the index structure used by our retrieval system (inverted files), explain Okapi BM25, one of the most popular relevance ranking techniques, and then present the structural query language GCL which can be used to express queries like

Find all documents in which “mad” and “cow” occur within a distance of 3 words from each other.

We also show how BM25 and GCL can be combined in order to compute collection term statistics on the fly. We chose GCL as the underlying query language because it offers very light-weight operators to define structural query constraints.

3.1 Index Structure: Inverted Files

Inverted files are the underlying index data structure in most information retrieval systems. An inverted file contains a set of *inverted lists* (also called *posting lists*). For each term in the index, its posting list contains the exact positions of all occurrences of this term in the text collection that the index refers to.

Techniques to efficiently construct an inverted file from a text collection and to maintain it (i.e., apply updates, such as document insertions and deletions, to the index) have been discussed elsewhere [HZ03, LZW04, BC05]. What is important is that the inverted index can be used to process search queries very efficiently. For every query term, its posting list is fetched from the index, and only the query terms’ posting lists are used to process the query. At query time, it is not necessary to actually look into the files that are being searched, as all the necessary information is stored in the index.

3.2 Relevance Ranking: TF/IDF and the Okapi BM25 Scoring Function

Most relevance ranking functions used in today’s information retrieval systems are based on the vector space model and the TF/IDF scoring paradigm. Other techniques, such as latent semantic indexing [DDL⁺90] or Google’s pagerank [PBMW98], do exist, but cannot be used for file system search:

- Latent semantic indexing is appropriate for information retrieval purposes, but not for the known-item search task associated with file system search; users are searching for the exact occurrence of query terms in files, not for semantic concepts.
- Pagerank cannot be used because there are usually no cross-references between the files in a file system.

Suppose a user sends a query to the search system requesting certain pieces of data matching the query (*files* in our scenario, *documents* in traditional information retrieval). The vector space model considers the query and all documents in the text collection (files in the file system) vectors in an n -dimensional vector space, where n is the number of different terms in the search system’s vocabulary (this essentially means that all terms are considered independent). The document vectors are ranked by their similarity to the query vector.

TF/IDF (*term frequency, inverse document frequency*) is one possible way to define the similarity between a document and the search query. It means that a document is more relevant if it contains more occurrences of a query term (has greater TF value), and that a query term is more important if it occurs in fewer documents (has greater IDF value). Many different TF/IDF scoring functions exist. One of the most prominent (and most sophisticated) is Okapi BM25 [RWJ⁺94] [RWHB98].

A BM25 query is a set of (T, q_T) pairs, where T is a query term and q_T is T ’s within-query weight. For example, when a user searches for “full text search” (not as a phrase, but as 3 individual terms), this results in the BM25 query

$$\mathcal{Q} := \{(\text{“full”}, 1), (\text{“text”}, 1), (\text{“search”}, 1)\}.$$

Given a query \mathcal{Q} and a document D , the document’s BM25 relevance score is:

$$s(D) = \sum_{(T, q_T) \in \mathcal{Q}} \frac{q_T \cdot w_T \cdot d_T \cdot (1 + k_1)}{d_T + k_1 \cdot ((1 - b) + b \cdot \frac{dl}{avgdl})}, \quad (1)$$

where d_T is the number of occurrences of the term T within D , dl is the length of the document D (number of tokens), and $avgdl$ is the average document length in the system. The free parameters are usually chosen as

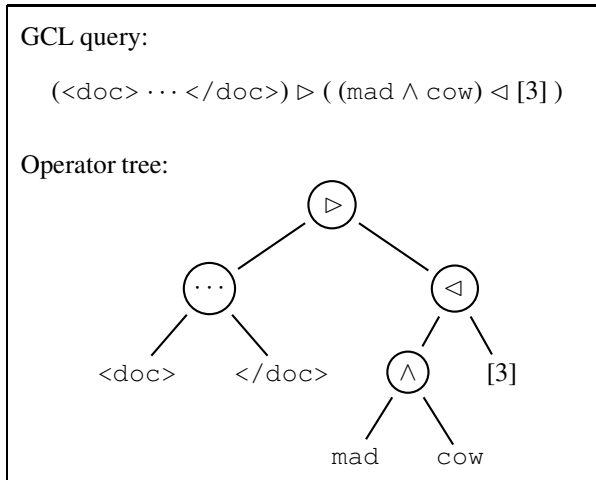


Figure 1: GCL query and resulting operator tree for the example query: *Find all documents in which “mad” and “cow” occur within a distance of 3 words from each other.* Leaf nodes in the operator tree correspond to posting lists in the index.

$k_1 = 1.2$ and $b = 0.75$. w_T is the IDF weight of the query term T :

$$w_T = \log\left(\frac{|\mathcal{D}|}{|\mathcal{D}_T|}\right), \quad (2)$$

where \mathcal{D} is the set of all documents in the text collection and \mathcal{D}_T is the set of all documents containing T . In our scenario, the documents are files, and we consequently denote \mathcal{D} as \mathcal{F} in the following sections.

From a Boolean point of view, a BM25 query is an OR query. Every document that contains at least one of the query terms matches the query. All matching documents are ranked according to their BM25 score. Roughly spoken (and therefore incorrect), this ranking makes documents containing all $|\mathcal{Q}|$ query terms appear at the top of the result list, followed by documents containing $|\mathcal{Q}| - 1$ different query terms, and so on.

3.3 Structural Queries: The GCL Query Language

The GCL (*generalized concordance lists*) query language proposed by Clarke et al. [CCB95] supports structural queries of various types. We give a brief introduction to the language because our safe implementation of the security model is based on GCL.

GCL assumes that the entire text collection (file contents) is indexed as a continuous stream of tokens. There is no explicit structure in this token stream. However, structural components, such as files or directories, can be added implicitly by inserting `<file>` and `</file>` tags (or `<dir>` and `</dir>`) into the token stream.

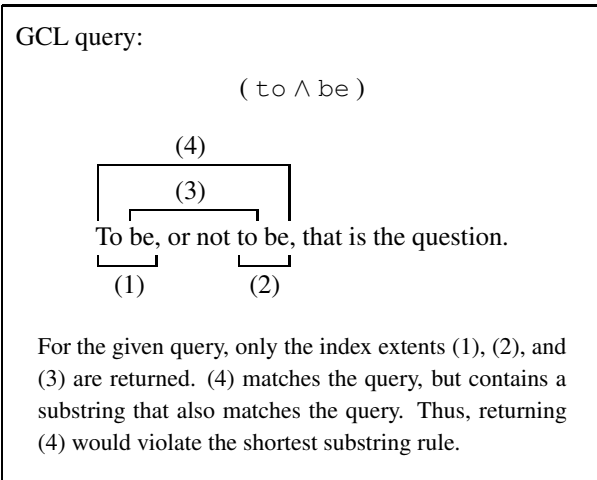


Figure 2: An example GCL query and the effect of the shortest substring rule: *Find all places where “to” and “be” co-occur.*

A GCL expression evaluates to a set of *index extents* (i.e., $[start, end]$ intervals of index positions). This is done by first producing an operator tree from the given GCL expression and then repeatedly asking the root node of the operator tree for the next matching index extent after the one that was seen last, until there are no more such extents.

GCL’s shortest substring paradigm demands that if two nested index extents satisfy a query condition, only the inner extent is part of the result set. This restriction limits the number of possible results to a query by the size of the text collection, whereas without it there could be $\binom{n}{2} \in \Theta(n^2)$ result extents for a text collection of size n . An example of how the application of the shortest substring rule affects the result to a GCL query is shown in Figure 2.

GCL operators are functions computing an output extent list from two or more input extent lists. This computation is performed on-demand in order to keep memory and CPU requirements low. The most basic type of extent lists are posting lists. As mentioned before, posting lists contain all occurrences of a given term within the indexed collection. They are found at the leaves of a GCL operator tree and can be combined using the various GCL operators.

The original GCL framework supports the following operators, which are only informally described here. Assume E is an index extent and A and B are GCL expressions. Then:

- E matches $(A \wedge B)$ if it matches both A and B ;
- E matches $(A \vee B)$ if it matches A or B ;
- E matches $(A \cdots B)$ if it has a prefix matching A and a suffix matching B ;

- E matches $(A \supset B)$ if it matches A and contains an extent E_2 matching B ;
- E matches $(A \not\supset B)$ if it matches A and does not contain an extent matching B ;
- E matches $(A \triangleleft B)$ if it matches A and is contained in an extent E_2 matching B ;
- E matches $(A \not\triangleleft B)$ if it matches A and is not contained in an extent matching B ;
- $[n]$ returns all extents of length n (i.e. all n -term sequences).

We have augmented the original GCL framework by two additional operators that let us compute collection statistics:

- $\#(A)$ returns the number of extents that match the expression A ;
- $\text{length}(A)$ returns the sum of the lengths of all extents that match A .

Throughout this paper, we use the same notation for both a GCL expression and the set of index extents that are represented by the expression.

3.4 BM25 and GCL

With the two new operators introduced in section 3.3, GCL can be employed to calculate all TF and IDF values that the query processor needs during a BM25 ranking process. Suppose the set of all documents inside the text collection is given by the GCL expression

`<doc>...</doc>`,

denoted as *documents*, and the document whose score is to be calculated is D . Then D 's relevance score is:

$$\sum_{T \in Q} w_T \cdot \frac{\#(T \triangleleft D) \cdot (1 + k_1)}{\#(T \triangleleft D) + k_1 \cdot ((1 - b) + b \cdot \frac{dl}{avgdl})}, \quad (3)$$

where

$$\begin{aligned} w_T &= \log \left(\frac{\#(documents)}{\#((documents) \supset T)} \right), \\ dl &= \text{length}(D), \text{ and} \\ avgdl &= \frac{\text{length}(documents)}{\#(documents)}. \end{aligned}$$

This is very convenient because it allows us to compute all collection statistics necessary to rank the search results on the fly. Thus, integrating the necessary security restrictions into the GCL query processor in such a way that no file permissions are violated, automatically guarantees consistent search results for relevance queries. We will use this property in section 6.

	<i>owner</i>	<i>group</i>	<i>others</i>
Read	x	x	
Write	x		
eXecute	x		x

Figure 3: File permissions in UNIX (example). *Owner* permissions override *group* permissions; *group* permissions override *others*. Access is either granted or rejected explicitly (in the example, a member of the group would not be allowed to execute the file, even though everybody else is).

4 A File System Search Security Model

In section 1, we have used the term *searchable* to refer to a file whose content is accessible by a user through the search system. In this section, we give a definition of what it means for a file to be searchable by user. Before we do so, however, we have to briefly revisit the UNIX security model, on which our security model is based, and discuss the traditional UNIX search paradigm.

The UNIX security model [RT74] [Rit78] is an ownership-based model with discretionary access control that has been adopted by many operating systems. Every file is owned by a certain user. This user (the file *owner*) can associate the file with a certain *group* (a set of users) and grant access permissions to all members of that group. He can also grant access permissions to the group of all users in the system (*others*). Privileges granted to other users can be revoked by the owner later on.

Extensions to the basic UNIX security model, such as access control lists [FA88], have been implemented in various operating systems (e.g., Windows, Linux), but the simple *owner/group/others* model is still the dominant security paradigm.

UNIX file permissions can be represented as a 3×3 matrix, as shown in Figure 3. When a user wants to access a file, the operating system searches from left to right for an applicable permission set. If the user is the owner of the file, the leftmost column is taken. If the user is not the owner, but member of the group associated with the file, the second column is taken. Otherwise, the third column is taken. This can, for instance, be used to grant read access to all users in the system except for those who belong to a certain group.

File access privileges in UNIX fall into three different categories: **Read**, **Write**, and **eXecute**. Write permissions can be ignored for the purpose of this paper, which does not deal with file changes. The semantics of the read and execute privileges are different depending on

whether they are granted for a file or a directory. For files,

- the read privilege entitles a user to read the contents of a file;
- the execute privilege entitles a user to run the file as a program.

For directories,

- the read privilege allows a user to read the directory listing, which includes file names and attributes of all files and subdirectories within that directory;
- the execute privilege allows a user to access files and subdirectories within the directory.

In the traditional `find/grep` paradigm, a file can only be searched by a user if

1. the file can be found in the file system's directory tree and
2. its content may be read by the user.

In terms of file permissions, the first condition means that there has to be a path from the file system's root directory to the file in question, and the user needs to have both the read privilege and the execute privilege for every directory along this path, while the second condition requires the user to have the read privilege for the actual file in question. The same rules are used by `slocate`⁸ to decide whether a matching file may be displayed to the user or not.

While these rules seem to be appropriate in many scenarios, they have one significant shortcoming: It is not possible to grant *search* permission for a single file without revealing information about other files in the same directory. In order to make a file searchable by other users, the owner has to give them the read privilege for the file's parent directory, which reveals file names and attributes of all other files within the same directory.

A possible solution to this problem is to relax the definition of *searchable* and only insist that there is a path from the file system root to the file in question such that the user has the execution privilege for every directory along this path. Unfortunately, this conflicts with the traditional use of the read and execution privileges, in which this constellation is usually used to give read permission to all users who know the exact file name of the file in question (note that, even without read permission for a directory, a user can still access all files in it; he just cannot use `ls` to search for them). While we think this not as big a problem as the make-the-whole-directory-visible problem above, it still is somewhat unsatisfactory.

The only completely satisfying solution would be the introduction of an explicit fourth access privilege, the

search privilege, in addition to the existing three. Since this is very unlikely to happen, as it would probably break most existing UNIX software, we base our definition of *searchable* on a combination of read and execute. A file is searchable by a user if and only if

1. there is a path from the root directory to the file such that the user has the *execute* privilege for all directories along this path and
2. the user has the *read* privilege for the file.

Search permissions can be granted and revoked, just as any other permission types, by modifying the respective read and execute privileges.

While our security model is based on the simple *owner/group/other* UNIX security model, it can easily be extended to other security models, such as access control lists, as long as the set of privileges (R, W, X) stays the same, because it only requires a user to have certain privileges and does not make any assumptions about where these privileges come from.

This is our basic security model. In order to fully implement this model, a search system must not deliver query results that depend on files that are not searchable by the user who submitted the query. Two possible implementations of the model are discussed in the following sections. The first implementation does not meet this additional requirement, while the second does.

While our implementation of the security model is based on the above definition of *searchability*, the security problems we are discussing in the following sections are independent of this definition. They arise in any environment in which there are files that may not be searched by a given user.

It should be noted at this point that in most UNIX file systems the content of a file is actually associated with an i-node instead of the file itself, and there can be multiple files referring to the same i-node. This is taken into account by our search engine by assuming an i-node to be searchable if and only if there is at least one hard link to that i-node such that the above rules hold for the link.

5 A First Implementation of the Security Model and How to Exploit It: The Postprocessing Approach

One possible implementation of the security model is based on the postprocessing approach described in section 1. Whenever a query is processed, system-wide term statistics (IDF values) are used to rank all matching files by decreasing similarity to the query. This is always done in the same way, regardless of which user sent the search query. After the ranking process has finished, all files

for which the user does not have search permission (according to the rules described in the previous section) are removed from the final list of results.

Using system-wide statistics instead of user-specific data suggests itself because it allows the search system to precompute and store all IDF values, which, due to storage space requirements, is not possible for per-user IDF values in a multi-user environment. Precomputing term statistics is necessary for various query optimization techniques [WL93] [PZSD96].

In this section, we show how this approach can be exploited to calculate (or approximate) the number of files that contain a given term, even if the user sending the query does not have read permissions for those files. Depending on whether the search system returns the actual BM25 file scores or only a ranked list of files, without any scores, it is either possible to compute exact term statistics (if scores returned) or approximate them (if no scores returned).

One might argue that revealing to an unauthorized user the number of files that contain a certain term is only a minor problem. We disagree. It is a major problem, and we give two example scenarios in which the ability to infer term statistics can have disastrous effects on file system security:

- An industrial spy knows that the company he is spying on is developing a new chemical process. He starts monitoring term frequencies for certain chemical compounds that are likely to be involved in the process. After some time, this will have given him enough information to tell which chemicals are actually used in the process – without reading any files.
- The search system can be used as a covert channel to transfer information from one user account to another, circumventing security mechanisms like file access logging.

Throughout this section we assume that the number of files in the system is sufficiently large so that the addition of a single file does not modify the collection statistics significantly. This assumption is not necessary, but it simplifies the calculations.

5.1 Exploiting BM25 Relevance Scores

Suppose the search system uses a system-wide index and implements Okapi BM25 to perform relevance ranking on files matching a search query. After all files matching a user's query have been ranked by BM25, all files that may not be searched by the user are removed from the list. The remaining files, along with their relevance scores, are presented to the user.

We will determine the total number of files in the file system containing the term T^* (the “*” is used to remind us that this is the term we are interested in). We start by computing the values of the unknown parameters $avgdl$ and $|\mathcal{F}|$ in the BM25 scoring function, as shown in equation (1). We start with $|\mathcal{F}|$, the number of files in the system. For this purpose, we generate two random terms T_2 and T_3 that do not appear in any file. We then create three files F_1 , F_2 , and F_3 :

- F_1 contains only the term T_2 ;
- F_2 consists of two occurrences of the term T_2 ;
- F_3 contains only the term T_3 .

Now, we send two queries to the search engine: $\{T_2\}$ and $\{T_3\}$. For the former, the engine returns F_1 and F_2 ; for the latter, it returns F_3 . For the scores of F_1 and F_3 , we know that

$$score(F_1) = \frac{(1 + k_1) \cdot \log(\frac{|\mathcal{F}|}{2})}{1 + k_1 \cdot ((1 - b) + \frac{b}{avgdl})} \quad (4)$$

and

$$score(F_3) = \frac{(1 + k_1) \cdot \log(\frac{|\mathcal{F}|}{1})}{1 + k_1 \cdot ((1 - b) + \frac{b}{avgdl})} \quad (5)$$

Dividing (4) by (5) results in

$$\frac{score(F_1)}{score(F_3)} = \frac{\log(\frac{|\mathcal{F}|}{2})}{\log(|\mathcal{F}|)} \quad (6)$$

and thus

$$|\mathcal{F}| = 2^{\frac{score(F_3)}{score(F_3) - score(F_1)}}. \quad (7)$$

Now that we know $|\mathcal{F}|$, we proceed and compute the only remaining unknown, $avgdl$. Using equation (5), we obtain

$$avgdl = \frac{b}{X - 1 + b}, \quad (8)$$

where

$$X = \frac{(1 + k_1) \cdot \log(|\mathcal{F}|) - score(F_3)}{score(F_3) \cdot k_1}. \quad (9)$$

Since now we know all parameters of the BM25 scoring function, we create a new file F_4 which contains the term T^* that we are interested in, and submit the query $\{T^*\}$. The search engine returns F_4 along with $score(F_4)$. This information is used to construct the equation

$$score(F_4) = \frac{(1 + k_1) \cdot \log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|})}{1 + k_1 \cdot ((1 - b) + \frac{b}{avgdl})}, \quad (10)$$

in which $|\mathcal{F}_{T^*}|$ is the only unknown. We can therefore easily calculate its value and after only two queries know the number of files containing T^* :

$$|\mathcal{F}_{T^*}| = |\mathcal{F}| \cdot \left(\frac{1}{2}\right)^{\text{score}(F_4) \cdot Y}, \quad (11)$$

where

$$Y = \frac{1 + k_1 \cdot ((1 - b) + \frac{b}{\text{avgdl}})}{1 + k_1}. \quad (12)$$

To avoid small changes in \mathcal{F} and avgdl , the new file F_4 can be created before the first query is submitted to the system.

While this particular technique only works for BM25, similar methods can be used to obtain term statistics for most TF/IDF-based scoring functions.

5.2 Exploiting BM25 Ranking Results

An obvious countermeasure against this type of attack is to restrict the output a little further and not return relevance scores to the user. We now show that even if the response does not contain any relevance scores, it is still possible to compute an approximation of $|\mathcal{F}_{T^*}|$, or even its exact value, from the order in which matching files are returned by the query processor. The accuracy of the approximation obtained depends on the number of files F_{\max} that a user may create. We assume $F_{\max} = 2000$.

The basic observation is that most interesting terms are infrequent. This fact is used by the following strategy: After we have created a single file F_0 , containing only the term T^* , we generate a unique, random term T_2 and create 1000 files $F_1 \dots F_{1000}$, each containing the term T_2 . We then submit the query $\{T^*, T_2\}$ to the search system. Since BM25 performs a Boolean OR to determine the set of matching files, F_0 as well as $F_1 \dots F_{1000}$ match the query and are ranked according to their BM25 score. If in the response to the query the file F_0 appears before any of the other files ($F_1 \dots F_{1000}$), we know that $|\mathcal{D}_{T^*}| \leq 1000$ and can perform a binary search, varying the number of files containing T_2 , to compute the exact value of $|\mathcal{F}_{T^*}|$.

If instead F_0 appears after the other files ($F_1 \dots F_{1000}$), at least we know that $|\mathcal{F}_{T^*}| \geq 1000$. It might be that this information is enough evidence for our purpose. However, if for some reason we need a better approximation of $|\mathcal{F}_{T^*}|$, we can achieve that, too.

We first delete all files we have created so far. We then generate a second random term T_3 and create 1,000 files ($F'_1 \dots F'_{1000}$), each containing the two terms T_2 and T_3 . We generate a third random term T_4 and create 999 files ($F'_{1001} \dots F'_{1999}$) each of which contains T_4 . We finally create a last file F'_0 that contains the two terms T^* and T_4 .

After we have created all the files, we submit the query $\{T^*, T_2, T_3, T_4\}$ to the search system. The relevance scores of the files $F'_0 \dots F'_{1000}$ are:

$$\text{score}(F'_0) = C \cdot (\log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}) + \log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_4}|})), \quad (13)$$

because F'_0 contains T^* and T_4 , and

$$\text{score}(F'_i) = C \cdot (\log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_2}|}) + \log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T_3}|})), \quad (14)$$

(for $1 \leq i \leq 1000$), because all the F'_i contain T_2 and T_3 . The constant C , which is the same for all files created, is the BM25 length normalization component for a document of length 2:

$$C = \frac{1 + k_1}{1 + k \cdot ((1 - b) + \frac{2 \cdot b}{\text{avgdl}})}. \quad (15)$$

We now subsequently delete one of the files $F'_{1001} \dots F'_{1999}$ at a time, starting with F'_{1999} , until $\text{score}(F'_0) \geq \text{score}(F'_1)$ (i.e. F'_0 appears before F'_1 in the list of matching files). Let us assume this happens after d deletions. Then we know that

$$\log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}) + \log(\frac{|\mathcal{F}|}{1000 - d}) \quad (16)$$

$$\geq 2 \cdot \log(\frac{|\mathcal{F}|}{1000}) \quad (17)$$

$$\geq \log(\frac{|\mathcal{F}|}{|\mathcal{F}_{T^*}|}) + \log(\frac{|\mathcal{F}|}{1000 - d + 1}) \quad (18)$$

and thus

$$-\log(|\mathcal{F}_{T^*}|) - \log(1000 - d) \quad (19)$$

$$\geq -2 \cdot \log(1000) \quad (20)$$

$$\geq -\log(|\mathcal{F}_{T^*}|) - \log(1000 - d + 1), \quad (21)$$

which implies

$$\frac{1000^2}{1000 - d} \geq |\mathcal{F}_{T^*}| \geq \frac{1000^2}{1000 - d + 1}. \quad (22)$$

If $|\mathcal{F}_{T^*}| = 11000$, for example, this technique would give us the following bounds:

$$10990 \leq |\mathcal{F}_{T^*}| \leq 11111.$$

The relative error here is about 0.5%. Again, binary search can be used to reduce the number of queries necessary from 1000 to around 10.

If it turns out that the approximation obtained is not good enough (e.g., if $|\mathcal{F}_{T^*}| > 40000$, we have a relative error of more than 2%), we repeat the process, this time with more than 2 terms per file. For $|\mathcal{F}_{T^*}| = 40001$ and 3 terms per file, for instance, this would give us the approximation

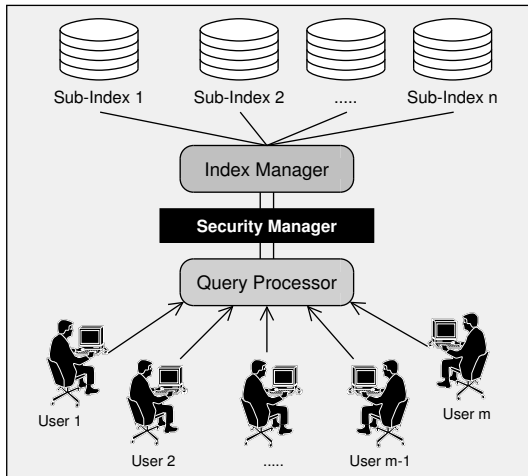


Figure 4: General layout of the Wumpus search system. The index manager maintains multiple sub-indices, one for every mount point. When the query processor requests a posting list, the index manager combines sub-lists from all indices into one large list and passes it to the security manager which applies user-specific security restrictions.

$$39556 \leq |\mathcal{F}_{T^*}| \leq 40057 \quad (\text{relative error: } 0.6\%)$$

instead of

$$40000 \leq |\mathcal{F}_{T^*}| \leq 41666 \quad (\text{relative error: } 2.1\%).$$

Thus, we have shown a way to systematically compute a very accurate approximation of the number of files in the file system containing a given term. Hence, if the postprocessing approach is taken to implement the search security model, it is possible for an arbitrary user to obtain information about the content of files for which he does not have read permission – by simply looking at the order in which files are returned by the search engine.

5.3 More Than Just Statistics

The above methods can be used to calculate the number of files that contain a particular term. While this already is undesirable, the situation is much worse if the search system allows relevance ranking for more complicated queries, such as boolean queries and phrases.

If, for instance, the search system allows phrase queries of arbitrary length, then it is possible to use the search system to obtain the whole content of a file. Assume we know that a certain file contains the phrase “A B C”. We then try all possible terms D and calculate the number of files which contain “A B C D” until we have found a D that gives a non-zero result. We then continue with the next term E . This way, it is possible to construct the entire content of a file using a finite number of search queries (although this might take a

long time). A simple n -gram language model [CGHH91] [GS95] can be used to predict the next word and thus increase the efficiency of this method significantly.

6 A Second Implementation of the Security Model: Query Integration

In this section, we describe how structured queries can be used to apply security restrictions to search results by integrating the security restrictions into the query processing instead of applying them in a postprocessing step. This implementation of the file system search security model is part of the Wumpus search system.

The general layout of the retrieval system is shown in Figure 4. A detailed description is given by [BC05]. All queries (Boolean and relevance queries) are handled by the query processor module. In the course of processing a query, it requests posting lists from the index manager. Every posting list that is sent back to the query processor has to pass the security manager, which applies user-specific restrictions to the list. As a result, the query processor only sees those parts of a posting list that lie within files that are searchable by the user who submitted the query. Since the query processor’s response is solely dependent on the posting lists it sees, the results are guaranteed to be consistent with the user’s view of the file system.

We now explain how security restrictions are applied within the security manager. In our implementation, every file in the file system is represented by an index extent satisfying the GCL expression

`<file> ... </file>.`

Whenever the search engine receives a query from a user U , the security manager is asked to compute a list F_U of all index extents that correspond to files whose content is searchable by U (using our security model’s definition of *searchable*). F_U represents the user’s view of the file system at the moment when the search engine received the query. Changes to the file system taking place while the query is being processed are ignored; the same list F_U is used to process the entire query.

While the query is being processed, every time the query processor asks for a term’s posting list (denoted as \mathcal{P}_T), the index manager generates \mathcal{P}_T and passes it to the security manager, which produces

$$\mathcal{P}_T^{(U)} \equiv (\mathcal{P}_T \triangleleft F_U),$$

the list of all occurrences of T within files searchable by U . The operator tree that results from adding these

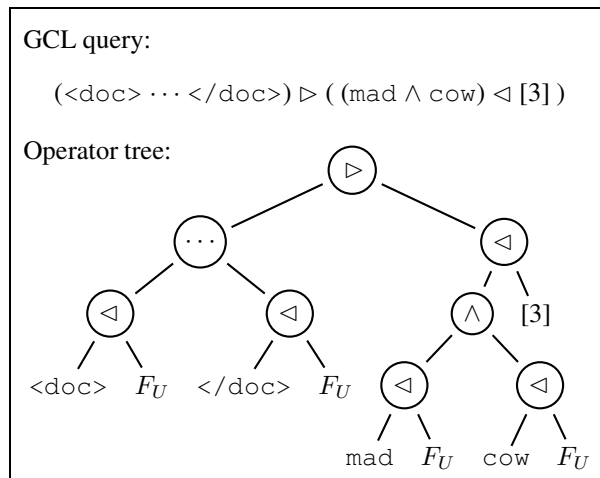


Figure 5: Integrating security restrictions into the query processing – GCL query and resulting operator tree with security restrictions applied to all posting lists (GCL containment operator “ \triangleleft ”).

security restrictions to a GCL query is shown in Figure 5. Their effect on query results is shown in Figure 6.

Since $\mathcal{P}_T^{(U)}$ is all the query processor ever sees, it is impossible for it to produce query results that depend on the content of files that are not searchable by U . Using the equations from section 3.4, all statistics necessary to perform BM25 relevance ranking can be generated from the user-specific posting lists, making it impossible to infer system-wide term statistics from the order in which matching files are returned to the user.

Because all operators in the GCL framework support lazy evaluation, it is not necessary to apply the security restrictions to the entire posting list when only a small portion of the list is used to process a query. This is important for query processing performance.

It is worth pointing out that this implementation of the security model has the nice property that it automatically supports index update operations. When a file is deleted from the file system, this file system change has to be reflected by the index immediately. Without a security model, every file deletion would either require an expensive physical update of the internal index structures, or a postprocessing step would be necessary in which all query results that refer to deleted files are removed from the final list of results [CH98]. The postprocessing approach would have the same problems as the one described in section 5: It would use term statistics that do not reflect the user’s actual view of the file system. With our implementation of the security model, file deletions are automatically supported because the

$\langle \text{file} \rangle \dots \langle / \text{file} \rangle$

extent associated with the deleted file is removed from

the security manager’s internal representation of the file system. This way, it is possible to keep the index up-to-date at minimal cost. Updates to the actual index data, which are very expensive, may be delayed and applied in batches. A more thorough discussion of index updates and their connection to security mechanisms is given by [BC05].

One drawback of our current implementation is that, in order to efficiently generate the list of index extents representing all files searchable by a given user, the security manager needs to keep some information about every indexed i-node in main memory. This information includes the i-nodes start and end address in the index address space, owner, permissions, etc. and comes to a total of 32 bytes per i-node. For file systems with a few million indexable files, this can become a problem. Keeping this information on disk, on the other hand, is not a satisfying solution, either, since it would make sub-second query times impossible. Unfortunately, we are not aware of a convincing solution to this problem.

7 Performance Evaluation

We evaluated the performance of both implementations of the security model – postprocessing and query integration – using a text collection known as TREC4+5-CR (TREC disks 4 and 5 without the Congressional Record). This collection contains 528,155 documents, which we split up into 528,155 different files in 5,282 directories. The index for this 2-GB text collection, with full positional information, requires about 615 MB, including 73 MB that are consumed by the search system’s internal representation of the directory tree, comprising file names for all files. The i-node table containing file access privileges has a total size of 16 MB and has to be kept in memory at all times to allow for fast query processing.

As query set, we used Okapi BM25 queries that were created by taking the 100 topics employed in the TREC 2003 Robust track and removing all stop words (using a moderately-sized set of 80 stop words). The original topics read like:

Identify positive accomplishments of the Hubble telescope since it was launched in 1991.

The topics were translated into queries that could be parsed and executed by our retrieval system:

```
@rank[bm25] "<doc>.."</doc>" by
  "positive", "accomplishments", "hubble",
  "telescope", "since", "launched", "1991"
```

On average, a query contained 8.7 query terms, which is significantly more than the 2.2 terms found in an average web search query [JSBS98]. Nonetheless, our system

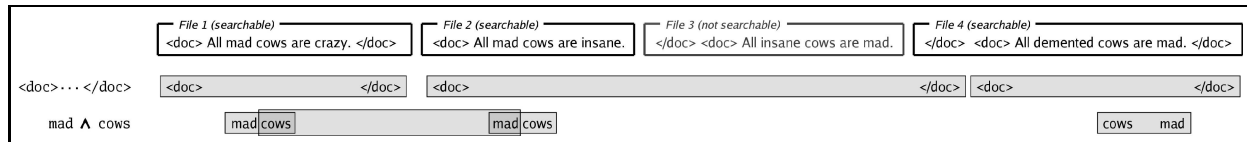
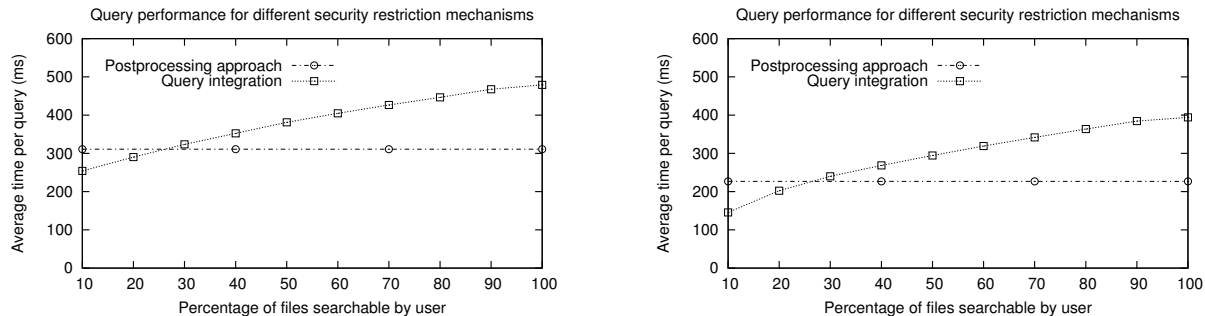


Figure 6: Query results for two example queries (“<doc>...</doc>” and “mad \wedge cows”) with security restrictions applied. Only postings from files that are searchable by the user are considered by the query processor.



(a) Without cache effects: All posting lists have to be fetched from disk.

(b) With cache effects: All posting lists are fetched from the disk cache.

Figure 7: Performance comparison – query integration using GCL operators vs. postprocessing approach. When the number of files searchable is small, query integration is more efficient than postprocessing because relevance scores for fewer documents have to be computed.

can execute the queries in well below a second on the TREC4+5-CR text collection used in our experiments.

We ran several experiments for different percentages of searchable files in the entire collection. This was done by changing the file permissions of all files in the collection between two experiments, making a random $p\%$ readable and the other files unreadable. This way, we are able to see how the relative number of files searchable by the user submitting the search query affects the relative performance of postprocessing and query integration.

All experiments were conducted on a PC based on an AMD Athlon64 3500+ with 2 GB of main memory and a 7,200-rpm SATA hard drive.

The results depicted in Figure 7 show that the performance of the second implementation (query integration) is reasonably close to that of the postprocessing approach. Depending on whether the time that is necessary to fetch the postings for the query terms from disk is taken into account or not, the slowdown is either 54% (Figure 7(a)) or 74% (Figure 7(b)) – when 100% of the files in the index are searchable by the user submitting the query. Performance figures for both the cached and the uncached case are given because, in a realistic environment, system behavior is somewhere between these two extremes.

As the number of searchable files is decreased, query processing time drops for the query integration approach, since fewer documents have to be examined and fewer

relevance scores have to be computed, but remains constant for the postprocessing approach. This is, because in the postprocessing approach, the only part of the query processing is the postprocessing, which requires very little time compared to running BM25 on all documents. As a consequence, query integration is 18%/36% (uncached/cached) faster than postprocessing when only 10% of the files are searchable by the user who submitted the query.

8 Query Optimization

Although our GCL-based implementation of the security model does not exhibit an excessively decreased performance, it is still noticeably slower than the postprocessing approach if more than 50% of the files can be searched by the user (22-30% slowdown when 50% of the files are searchable). The slowdown is caused by applying the security restrictions ($\dots \triangleleft F_U$) not only to every query term but also to the document delimiters (<doc> and </doc>). Obviously, in order to guarantee consistent query results, it is only necessary to apply them to either the documents (in which case the query BM25 function will ignore all occurrences of query terms that lie outside searchable documents) or the query terms (in which case unsearchable documents would not contain any query terms and therefore receive

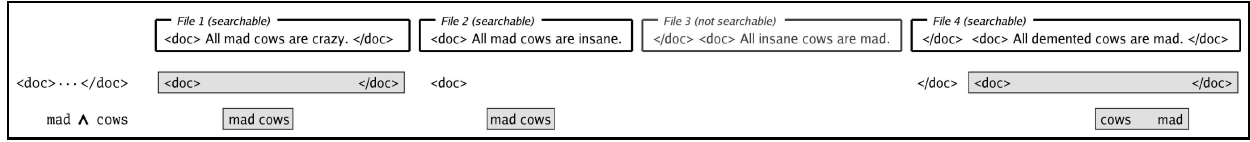


Figure 8: Query results for two example queries (“<doc> ... </doc>” and “mad & cows”) with revised security restrictions applied. Even though <doc> in file 2 and </doc> in file 4 are visible to the query processor, they are not considered valid query results, since they are in different files.

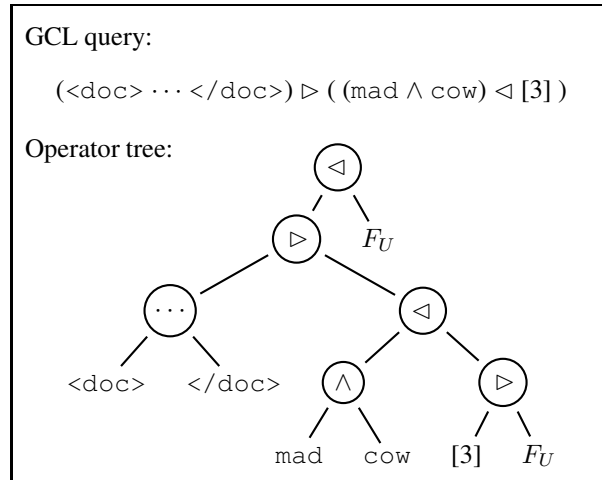


Figure 9: GCL query and resulting operator tree with optimized security restrictions applied to the operator tree.

a score of 0).

However, this optimization would be very specific to the type of the query (TF/IDF relevance ranking). More generally, we can see the equivalence of the following three GCL expressions:

$$\begin{aligned} ((E_1 \triangleleft F_U) \triangleright (E_2 \triangleleft F_U)), \\ ((E_1 \triangleleft F_U) \triangleright E_2), \text{ and} \\ (E_1 \triangleright E_2) \triangleleft F_U, \end{aligned}$$

where the first expression is the result of the implementation from section 6 when run on the GCL expression

$$(E_1 \triangleright E_2).$$

The three expressions are equivalent because if an index extent E is contained in another index extent E' , and E' is contained in a searchable file, then E has to be contained in a searchable file as well.

If we make the (not very constrictive assumption) that every index extent produced by one of the GCL operators has to lie completely within a file searchable by the user that submitted the query, then we get additional equivalences:

$$((E_1 \triangleleft F_U) \wedge (E_2 \triangleleft F_U)) \equiv ((E_1 \wedge E_2) \triangleleft F_U),$$

$$\begin{aligned} ((E_1 \triangleleft F_U) \vee (E_2 \triangleleft F_U)) &\equiv ((E_1 \vee E_2) \triangleleft F_U), \\ ((E_1 \triangleleft F_U) \cdots (E_2 \triangleleft F_U)) &\equiv ((E_1 \cdots E_2) \triangleleft F_U), \end{aligned}$$

and so on. Limiting the list of extents returned by a GCL operator to those extents that lie entirely within a single file conceptually means that all files are completely independent. This is not an unrealistic assumption, since index update operations may be performed in an arbitrary order when processing events associated with changes in the file system. Thus, no ordering of the files in the index can be guaranteed, which renders extents spanning over multiple files somewhat useless. The effect that this assumption has on the query results is shown in Figure 8.

Note that if we did not make the file independence assumption, then the right-hand side of the above equivalences would be more restrictive than the left-hand side (in the case of “ \vee ”, for example, the right-hand side mandates that both extents lie within the same searchable file, whereas the left-hand side only requires that both extents lie within searchable files). If we make the assumption, then in all the cases shown above we can freely decide whether the security restrictions should be applied at the leaves of the operator tree or whether they should be moved up in the tree in order to achieve better query performance.

The only GCL operator that does not allow this type of optimization is the *contained-in* operator. The GCL expression

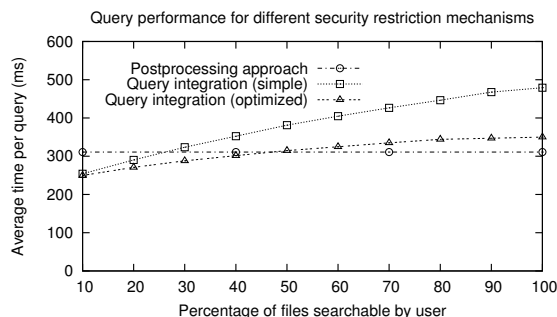
$$((E_1 \triangleleft F_U) \triangleleft (E_2 \triangleleft F_U))$$

is *not* equivalent to

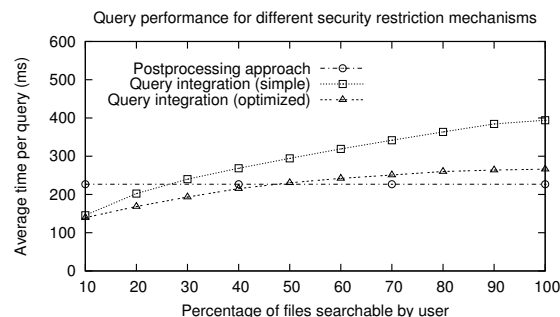
$$(E_1 \triangleleft E_2) \triangleleft F_U,$$

since in the second expression E_2 can refer to something outside the searchable files without the security restriction operator ($\triangleleft F_U$) “noticing” it. This would allow a user to infer things about terms outside the files searchable by him, so we cannot move the security restrictions up in the operator tree in this case.

At this point, it is not clear to us which operator arrangement leads to optimal query processing performance. Therefore, we follow the simple strategy of moving the security restrictions as far to the top of the operator tree as possible, as shown in Figure 9. Note that,



(a) Without cache effects: All posting lists have to be fetched from disk.



(b) With cache effects: All posting lists are fetched from the disk cache.

Figure 10: Performance comparison – query integration using GCL operators (simple and optimized) vs. postprocessing approach. Time per query for the optimized integration is between 61% and 117% compared to postprocessing.

in the figure, security restrictions are applied to the sub-expression “[3]” (all index extents of length 3), which, of course, does not make much sense, but is done by our implementation anyway.

Despite the possibility of other optimization strategies leading to better performance for certain queries, the move-to-top strategy works very well for “flat” relevance queries, such as the ones we used in our experiments:

$$(\langle \text{doc} \rangle \dots \langle / \text{doc} \rangle) \triangleright (T_1 \vee T_2 \vee \dots \vee T_n),$$

where the T_i are the query terms (remember that BM25 performs a Boolean OR). The performance gains caused by moving the security restrictions to the top of the tree are shown in Figure 10. With optimizations, the query integration is between 12-17% slower (100% files visible) and 20-39% faster (10% files visible) than the postprocessing approach. Even if most files in the file system are searchable by the user, this is only a minor slowdown that is probably acceptable, given the increased security.

9 Conclusion

Guided by the goal to reduce the overall index disk space consumption, we have investigated the security problems that arise if, instead of many per-user indices, a single system-wide index is used to process search queries from all users in a multi-user file system search environment.

If the same system-wide index is accessed by all users, appropriate mechanisms have to be employed in order to make sure that no search results violate any file permissions. Our full-text search security model specifies what it means for a user to have the privilege to search a file. It integrates into the UNIX security model and defines the search privilege as a combination of read and execution privileges.

For one possible implementation of the security model, based on the postprocessing approach, we have demonstrated how an arbitrary user can infer the number of files in the file system containing a given term, without having read access to these files. This represents a major security problem. The second implementation we presented, a query integration approach, does not share this problem, but may lead to a query processing slowdown of up to 75% in certain situations.

We have shown that, using appropriate query optimization techniques based on certain properties of the structural query operators in our retrieval system, this slowdown can be decreased to a point at which queries are processed between 39% faster and 17% slower by the query integration than by the postprocessing approach, depending on what percentage of the files in the file system is searchable by the user.

References

- [BC05] Stefan Büttcher and Charles L. A. Clarke. Indexing Time vs. Query Time Trade-offs in Dynamic Information Retrieval Systems. Wumpus Technical Report 2005-01. <http://www.wumpus-search.org/papers/wumpus-tr-2005-01.pdf>, July 2005.
- [BCC94] Eric W. Brown, James P. Callan, and W. Bruce Croft. Fast Incremental Indexing for Full-Text Information Retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192–202, Santiago, Chile, September 1994.
- [BJS95] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Database Security: Research and Practice. *Information Systems*, 20(7):537–556, 1995.
- [BSJ97] Elisa Bertino, Pierangela Samarati, and Sushil Jajodia. An Extended Authorization Model for Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):85–101, 1997.
- [CCB95] Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An Algebra for Structured Text Search and a

- Framework for its Implementation. *The Computer Journal*, 38(1):43–56, 1995.
- [CGHH91] Kenneth Church, William Gale, Patrick Hanks, and Donald Hindle. Using Statistics in Lexical Analysis. In Uri Zernik, editor, *Lexical Acquisition: Exploiting On-Line Resources to Build a Lexicon*, pages 115–164, 1991.
- [CH98] Tzi-cker Chiueh and Lan Huang. Efficient Real-Time Index Updates in Text Retrieval Systems. Technical report, Stony Brook University, Stony Brook, New York, USA, August 1998.
- [DDL⁺90] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by Latent Semantic Analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.
- [FA88] G. Fernandez and L. Allen. Extending the UNIX Protection Model with Access Control Lists. In *Proceedings of USENIX*, 1988.
- [Fag78] Ronald Fagin. On an Authorization Mechanism. *ACM Transactions on Database Systems*, 3(3):310–319, 1978.
- [GS95] William Gale and Geoffrey Sampson. Good-Turing Frequency Estimation Without Tears. *Journal of Quantitative Linguistics*, 2:217–237, 1995.
- [GW76] Patricia P. Griffiths and Bradford W. Wade. An Authorization Mechanism for a Relational Database System. *ACM Transactions on Database Systems*, 1(3):242–255, 1976.
- [HZ03] Steffen Heinz and Justin Zobel. Efficient Single-Pass Index Construction for Text Databases. *Journal of the American Society for Information Science and Technology*, 54(8):713–729, 2003.
- [JSBS98] Bernard J. Jansen, Amanda Spink, Judy Bateman, and Tefko Saracevic. Real Life Information Retrieval: A Study of User Queries on the Web. *SIGIR Forum*, 32(1):5–17, 1998.
- [LZW04] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-Place versus Re-Build versus Re-Merge: Index Maintenance Strategies for Text Retrieval Systems. In *Proceedings of the 27th Conference on Australasian Comp. Sci.*, pages 15–23. Australian Computer Society, Inc., 2004.
- [PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.
- [Pel97] Kyle Peltonen. Adding Full Text Indexing to the Operating System. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE 1997)*, pages 386–390. IEEE Computer Society, 1997.
- [PZSD96] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society for Information Science*, 47(10):749–764, October 1996.
- [Rit78] Dennis M. Ritchie. The UNIX Time-Sharing System: A Retrospective. *Bell Systems Technical Journal*, 57(6):1947–1969, 1978.
- [RT74] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974.
- [RWHB98] Stephen E. Robertson, Steve Walker, and Micheline Hancock-Beaulieu. Okapi at TREC-7. In *Proceedings of the Seventh Text REtrieval Conference (TREC 1998)*, November 1998.
- [RWJ⁺94] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *Proceedings of the Third Text REtrieval Conference (TREC 1994)*, November 1994.
- [TF95] Howard Turtle and James Flood. Query Evaluation: Strategies and Optimization. *Information Processing & Management*, 31(1):831–850, November 1995.
- [WL81] Paul F. Wilms and Bruce G. Lindsay. A Database Authorization Mechanism Supporting Individual and Group Authorization. In *Distributed Data Sharing Systems*, pages 273–292, 1981.
- [WL93] Wai Y. P. Wong and Dik L. Lee. Implementations of Partial Document Ranking Using Inverted Files. *Information Processing & Management*, 29(5):647–669, 1993.

Notes

¹<http://desktop.google.com/>

²<http://toolbar.msn.com/>

³<http://www.apple.com/macosx/features/spotlight/>

⁴<http://desktop.yahoo.com/>

⁵<http://www.copernic.com/>

⁶http://images.apple.com/macosx/pdf/MacOSX_Spotlight_TB.pdf

⁷<http://www.wumpus-search.org/>

⁸<http://www.geekreview.org/slocate/>

Matrix Methods for Lost Data Reconstruction in Erasure Codes

James Lee Hafner, Veera Deenadhayalan, and KK Rao
IBM Almaden Research Center

John A. Tomlin*
Yahoo! Research

hafner@almaden.ibm.com, [veerad,kk rao]@us.ibm.com tomlin@yahoo-inc.com

Abstract

Erasures codes, particularly those protecting against multiple failures in RAID disk arrays, provide a code-specific means for reconstruction of lost (erased) data. In the RAID application this is modeled as loss of strips so that reconstruction algorithms are usually optimized to reconstruct entire strips; that is, they apply only to highly correlated sector failures, i.e., sequential sectors on a lost disk. In this paper we address two more general problems: (1) recovery of lost data due to scattered or uncorrelated erasures and (2) recovery of partial (but sequential) data from a single lost disk (in the presence of any number of failures). The latter case may arise in the context of host IO to a partial strip on a lost disk. The methodology we propose for both problems is completely general and can be applied to any erasure code, but is most suitable for XOR-based codes.

For the scattered erasures, typically due to hard errors on the disk (or combinations of hard errors and disk loss), our methodology provides for one of two outcomes for the data on each lost sector. Either the lost data is declared unrecoverable (in the information-theoretic sense) or it is declared recoverable *and* a formula is provided for the reconstruction that depends only on readable sectors. In short, the methodology is both complete and constructive.

1 Introduction

XOR-based erasures codes for disk arrays model lost data most coarsely as loss of entire disks but more precisely as loss of entire symbols of the code. In practice, a symbol typically maps to a “strip”, that is, multiple sequential sectors with one bit of the symbol corresponding to one or (typically) more sectors and with each different symbol residing on a different disk (this is not always the case, but it is a common practice). The collection of related strips is called a “stripe”. To deal with disk failures, each erasure code comes complete with a specific reconstruction algorithm that is highly dependent on the code construction. For example, the 2-fault-tolerant X-code [10] is constructed geometrically, with parity values computed along diagonal paths through the

data sectors. When two disks fail, the reconstruction follows these diagonal paths, starting at some initial point; that is, the reconstruction is both geometrically and recursively defined. The BCP [1] code is less geometrically designed, but still has a recursive reconstruction algorithm. More examples are mentioned in Section 2.

Erasures then are seen as correlated sector failures: all sectors in a strip are “lost” when the disk fails. However, increasing disk capacity together with a fairly stable bit-error rate implies that there is a significant probability of multiple uncorrelated or scattered *sector* errors within a given stripe, particularly in conjunction with one or more disk failures. For example, two disk losses plus a sector loss may occur often enough that even a two-disk fault tolerant code may not provide sufficient reliability. If all correlated and uncorrelated erasures occur within at most t disks where t is the (disk) fault tolerance of the code, then one method is to simulate loss of all affected disks and rebuild according to the code-specific reconstruction algorithm. However, this has two drawbacks. First, it is clear that this can be highly inefficient since it requires either reconstruction of “known” or readable data or it requires checking at each step of the process to see if a reconstruction is required. More importantly, however, this approach does not solve the more general problem when *more* than t disks have been affected with sector losses. In such a case, it is quite possible that some or all of the lost sectors can be reconstructed, though this is not obvious *a priori* from the erasure correcting power of the code. For example, the 2-fault tolerant EVENODD code only claims to recover from two lost disks, so any additional sector loss typically means all lost data is declared unrecoverable. In fact, on average, anywhere from 40-60% of the lost sectors may be recovered in this situation.

In addition, while each erasure code provides a means to reconstruct entire strips (e.g., during a rebuild operation), to our knowledge, the literature does not contain any methods that explicitly address the problem of reconstructing a partial strip of lost data; such a need may arise in a host read operation to a failed disk during an incomplete rebuild operation. Of course, the strip reconstruction method could be applied in this case, but it is likely that such reconstruction will recover additional unnecessary lost sectors; that is, do more work than is

*This work was done while Dr. Tomlin was at the IBM Almaden Research Center

required to service the host read, thereby adversely affecting performance. (This extra work may be worth the performance penalty in that the additional recovered sectors can be cached or added to the rebuild log, but that may not always be a desirable option.)

In this paper, we address both these problems. Our methodology is universal in that it can be applied to any erasure code of any fault tolerance. It applies to any failure scenario from full disk to scattered sectors to combinations of the two. It is based solely on the generator matrix for the erasure code. Consequently, a general erasure code reconstruction module could implement this methodology and use the generator matrix as one of its inputs. To emphasize the point, we address the problem of arbitrary sector (bit) erasures for *any* code designed with a strip (symbol) erasure failure model.

For the first problem of scattered (correlated and/or uncorrelated) sector loss, our methodology provides a mathematical guarantee: for each lost sector, either that sector's data is declared as (information-theoretically) unrecoverable (that is, a "data loss event") *or* the sector's data is declared recoverable and a reconstruction formula is generated. The reconstruction formula is a linear equation (XOR equation in case of XOR-based codes) involving known or readable data and parity sectors. In this respect, our methodology is both complete, constructive and universally applicable. It provides the best guarantee to meet the following requirement:

User Contract: *For any erasure scenario, the storage system shall recover any and all lost data sectors that the erasure code is information-theoretically capable of recovering.*

It should be noted that for 1-fault tolerant codes (e.g., RAID1, RAID4 or RAID5), the solution to both these problems is quite simple and obvious. Similarly, for Reed-Solomon codes [9] where the symbol is mapped to bytes or words (not sets of sectors), the standard Reed-Solomon procedure addresses both problems directly as well. The more interesting cases then are non-Reed-Solomon multiple fault-tolerant codes. Such codes are typically XOR-based as those have the most practical application. Careful and complex analysis of a specific code may produce a solution to this problem (and to our second problem). However, our solutions are universal. It is also clear that our methods can be extended to more general codes (e.g., some of the non-XOR codes in [3]). Furthermore, this methodology can be applied not just for RAID controllers but any application of these types of erasure codes such as dRAID (distributed Redundant Arrangement of Independent Devices) node-based systems.

For the second problem of partial strip reconstruction, we propose a hybrid solution: combine the inherent recursive method of the erasure code for full rebuild with

the methodology for recovering scattered sectors. We also propose an alternative that is in many cases equivalent to the code-specific method, better in some cases and universally applicable to any erasure code.

Our methodology is based on principles of matrix theory and pseudo-inverses. Many codes (see [8, 9]) use full inverses to prove both that their codes have the declared fault tolerance and to provide reconstruction formulas. However, they apply it only to recover full code symbols, under maximal failures (where unique inverses exist) and not to the more general bit-within-a-symbol (*a.k.a.*, sector within a strip) level that we address in this work.

The paper is organized as follows. We close the introduction with some definitions. The next section contains a few remarks on related work. Section 3 contains a brief review of the concepts from linear algebra that we need, particularly the notion of pseudo-inverse. In Section 4 we present a brief description of the generator matrix and parity check matrix for an erasure code. Section 5 explains how we simulate scattered sector loss and how we determine reconstructability in addressing our first problem. Section 6 contains algorithms for constructing pseudo-inverse matrices. We develop our methods in a detailed example in Section 7. Section 8 outlines the hybrid method for partial strip reconstruction (our second problem) and includes experimental results. We conclude with a brief summary.

1.1 Vocabulary

sector: the smallest unit of IO to/from a disk (typically 512 bytes at the disk drive, but perhaps 4KB from the filesystem or application layer).

element: a fundamental unit of data or parity; this is the building block of the erasure code. In coding theory, this is the data that is assigned to a bit within a symbol. We assume for simplicity that each element corresponds to a single sector; the more general case can be derived from this case.

stripe: a complete (connected) set of data and parity elements that are dependently related by parity computation relations. In coding theory, this is a code word; we use "code instance" synonymously.

strip: a unit of storage consisting of all contiguous elements (data, parity or both) from the same disk and stripe. In coding theory, this is associated with a code symbol. It is sometimes called a stripe unit. The set of strips in a code instance form a stripe. Typically, the strips are all of the same size (contain the same number of elements).

array A collection of disks on which one or more instances of a RAID erasure code is implemented.

2 Related Work

The two main results of this paper are (a) the application of pseudo-inverses of matrices to the problem of reconstruction of uncorrelated lost sectors and (b) a hybrid reconstruction that combines code-specific recursive reconstruction methods with this matrix method to efficiently reconstruct partial strips. To our knowledge neither of these problems has been specifically addressed in the literature. As remarked before, the theory of matrix inverses is used in the proof that some codes meet their declared strip (i.e., symbol) fault tolerance. For example, the Reed-Solomon code [8, 9] proves fault tolerance by solving a system of linear equations. In this case, the matrix inverse method is used to solve for complete symbols (full strips in our terminology) under maximum failures. In contrast, our method addresses individual bits in symbols (i.e., elements) for any distribution of erased bits (within or beyond symbol fault tolerance). The binary BR [3] codes have a recursive solution to two full strip losses; the authors provide a closed form solution to the recursion. For the EVENODD code [2], the authors give a recursion and point out that it could be solved explicitly. An explicit solution to the recursion is equivalent to our matrix solution in the special case of full strip losses (again, our method has no such correlation requirements). The BCP [1], ZZS [11], X-code [10], and RDP [4] codes all have recursive reconstruction algorithms. The latter two (as well as the EVENODD code) are “geometric” and easy to visualize; the former are more “combinatorial” and less intuitive. In either case, these codes with recursive reconstruction algorithms are well-suited to our hybrid methodology. In addition, a variant of our hybrid method applies to any erasure codes suitable for disk arrays, with or without a recursive reconstruction algorithm.

3 Binary Linear Algebra – A Review

In this section we recall and elaborate on some basic notions from the theory of linear algebra over a binary field (which is assumed for all operations from now on without further comment – the theory extends easily to non-binary fields as well). A set of binary vectors is linearly independent if no subset sums modulo 2 to the zero vector. Let G be a rectangular matrix of size $N \times M$ with $N \leq M$. The “row rank” of G is the maximum number of linearly independent row vectors. The matrix G has “full row rank” if the row rank equals N (the number of rows). A “null space” for G is the set of *all* vectors that are orthogonal (have zero dot-product) with *every* row vector of G . This is a vector space closed under vector addition modulo 2. A “null space basis” is a maximal set of linearly independent vectors from the null space. If the null space basis has Q vectors, then the entire null space has $2^Q - 1$ total non-zero vectors.

We will write the null space vectors as column vectors, to make matrix multiplication simpler to write down, though this is not the standard convention.

Let B be a basis for the null space of G . More precisely, B is a matrix whose columns form a basis for the null space. If G has full row rank, then B has dimensions $M \times Q$ where $Q = M - N$.

Suppose G is full row rank. A “right pseudo-inverse” is a matrix R (of size $M \times N$) so that

$$G \cdot R = I_N$$

where I_N is the $N \times N$ identity matrix. If $M = N$, then R is the *unique* inverse. A right pseudo-inverse must exist if G has full rank and is never unique if $N < M$.

More generally, let G have row rank $K \leq N$, then a “partial right pseudo-inverse” (or partial pseudo-inverse) is a matrix R so that

$$G \cdot R = J_K$$

where J_K is an N -dimensional square matrix with K ones on the diagonal, $N - K$ zeros on the diagonal and zeros elsewhere. Note that R is a partial pseudo-inverse if the product $G \cdot R$ has a maximal number of ones over all possible choices for R . If G is full row rank then $K = N$, $J_N = I_N$ and R is a (complete) pseudo-inverse. The matrix J_K is unique; that is, the positions of zero and non-zero diagonal elements are determined from G and are independent of the choice of R .

Let B be a $M \times Q$ basis for the null space basis for G (perhaps padded with all-zero columns), and R some *specific* partial pseudo-inverse for G . As X varies over all binary $Q \times N$ matrices, we have

$$G \cdot (R + (B \cdot X)) = J_K. \quad (1)$$

so $R + (B \cdot X)$ runs over all partial pseudo-inverses (the proof of this is a simple calculation). What X does in (1) is add a null space vector to each of the columns of R . For our purposes, an optimal R would have minimum weight (fewer ones) in each column (that is, be the most sparse). In Section 6 we discuss algorithms for computing pseudo-inverses and in Section 6.2 we discuss algorithms for finding an optimal pseudo-inverse.

For each column of J_K with a zero on the diagonal, the corresponding column of R can be replaced with the all-zero column without affecting the partial pseudo-inverse property and in fact such an action clearly improves the weight of R . Consequently, we add this property to the definition of a partial pseudo-inverse.

Strictly speaking, the term “pseudo-inverse” applies only to real or complex matrices and implies uniqueness (optimality in a metric sense). We overload the term here with a slightly different meaning – we allow for non-uniqueness and do not require optimality (most sparse).

In the next section we apply these notions to the problem of reconstruction of scattered sectors in a stripe.

4 Generator and Parity Check Matrices

In this section we recall the erasure code notions of “generator matrix” and “parity check matrix”. These are the basic structures upon which we develop our methodology. For a basic reference, see [7].

The generator matrix G of an erasure code converts the input “word” (incoming data) into a “code word” (data and parity). The parity check matrix verifies that the “code word” contains consistent data and parity (parity scrub). In the context of erasure codes for disk arrays, the generator matrix actually provides much more.

The generator matrix is given a column block structure: each block corresponds to a strip and each column within a block corresponds to an element within the strip. If the column contains only a single 1, then the element contains user data. We call such a column an “identity column” because it is a column of an identity matrix. If the column contains multiple 1s, then it corresponds to an element which is the XOR sum of some set of user data elements; that is, the element is a parity element. In other words, the generator matrix specifies the data and parity layout on the strips, the logical ordering of the strips within the stripe, *and* the equations used to compute parity values. For example, the generator matrix for the EVENODD(3,5) code with prime $p = 3$ on 5 disks is

$$G = \left(\begin{array}{cc|cc|cc|cc|cc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array} \right).$$

(more details on this example are given in Section 7).

Though it is not a requirement, the generator matrix for disk arrays typically has an identity column for each user data element (so that this data is always copied to the element’s sectors verbatim in some strip and can then be read with minimal IO costs). In coding theory, a generator matrix of this form is called “systematic”.

Let D be a row vector of input user data values, then the row vector S , given by the expression

$$S = D \cdot G, \quad (2)$$

represents the data and parity elements that are stored in the stripe on the disks. The vector D is indexed by the logical addresses of the user data values (say, as viewed by the host). The vector S represents the physical addresses of the data and parity elements, both the disk (actually, strip, identified by the block of the generator matrix) and the sector addresses on the disk (element or

offset within the strip, identified by the column within the block). S is also block-structured with blocks matching those of G . (See our example in Section 7.)

If there are N data elements input into the code and Q parity elements computed by the code, then the generator matrix has dimensions $N \times (N + Q)$. (Note that N is the total number of data elements within a stripe, not the number of strips; similarly, Q is the number of parity elements in the stripe, not the number of parity strips.)

The “parity check matrix” H has dimensions $(N + Q) \times Q$ and can be derived directly from the generator matrix (and vice-versa). Communication channels use the parity check matrix to detect errors. Each column corresponds to a parity element. After the data and parity is read off the channel, the parity is XORed with the data as indicated by its corresponding column to produce a “syndrome”. If a syndrome is not zero, an error has occurred (either in the received parity symbol or in one of the dependent data symbols). For erasure codes in disk arrays, this is a parity consistency check (or parity scrub). In other words, with $S = D \cdot G$ as above, the test

$$S \cdot H = 0 \quad (3)$$

is a parity consistency check.

The parity check matrix is row blocked exactly corresponding to the column blocks of G (or S) and it can be arranged to contain an embedded identity matrix (corresponding to the parity elements) – this is easy if G is systematic. The parity check matrix for the example generator matrix G above is

$$H = \left(\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array} \right).$$

In short, the generator matrix is used to compute the data and parity (and its layout) for storage on the disks. The parity check matrix can be used when all the data and parity are read off the disk (e.g., during parity scrub) to look for errors.

If a code can tolerate $t \geq 1$ lost disks or strips, then G must have the property that if any t blocks of G are removed (or zeroed), then the resulting matrix must have full row rank. The parity check matrix is full column rank (because of the embedded identity matrix).

Also, (3) implies that

$$D \cdot G \cdot H = 0$$

should hold for every data vector D . This means that $G \cdot H = 0$ identically, so that each vector in H is in the null space of G . A simple dimensionality argument shows that in fact H is a basis of the null space of G .

In addition, it should be clear that if G is systematic, then there exists an $M \times N$ matrix R_0 containing an embedded identity matrix of size $N \times N$ so that R_0 is a pseudo-inverse for G . R_0 just picks off the embedded systematic portion of G . If G is not systematic, a pseudo-inverse R_0 can still be constructed, but it will not be so simple (see Section 6.3).

5 Simulating Scattered Sector Loss and Reconstruction

In this section, we develop our theory for solving the first of our two problems: how to deal with uncorrelated sector loss. An example is given in Section 7

We indicated above that a t -fault-tolerant code G must have the property that zeroing any t blocks of G should leave G full rank so that a complete pseudo-inverse for G must exist. This suggests that we can simulate correlated and/or uncorrelated sector loss by zeroing or removing the associated *individual* columns from G . It should be clear that certain combinations of uncorrelated sector losses will result in some or all data loss events (some or all lost sectors having unrecoverable data); other combinations may involve no data loss events. Our methodology will determine, in a straightforward manner, exactly what sectors become data loss events and for those that do not, will provide a reconstruction formula for the data from these sectors.

Suppose we detect a set F of failed sectors in a stripe (correlated, perhaps because of disk failure, or uncorrelated, because of medium errors, or a combination of these). Completely ignoring the block structure of G , let \hat{G} be a version of a generator matrix G , with zeroed columns corresponding to the sectors in F . Suppose we can find a matrix R of size $M \times N$ so that

- R is a partial pseudo-inverse of \hat{G} , and
- R has zeros in all rows that correspond to the lost columns of \hat{G} .

We associate the columns of R to the user data values in D . In Section 6 we discuss algorithms for constructing R . The following theorem contains our main theoretical result:

Theorem 1. *Let G , \hat{G} , and R be as above. Any theoretically recoverable user data value corresponds to a non-zero column of R and the non-zero bit positions indicate the data and parity elements whose XOR sum equals the data value. As a special case, a directly readable data value corresponds to an identity column in R . A data loss event (unrecoverable data value) corresponds to an all-zero column of R .*

Proof. Let \hat{S} be the vector S as in (2) but with zeros in the positions corresponding to the lost elements (the zeroed columns of G). Then it is clear that

$$D \cdot \hat{G} = \hat{S}.$$

Consequently, we have

$$\hat{S} \cdot R = D \cdot \hat{G} \cdot R = D \cdot J_K = \hat{D},$$

where \hat{D} is the vector D with zeros in all locations corresponding to zero's on the diagonal of J_K which also corresponds to the all-zero columns of R .

The fact that J_K is uniquely determined by \hat{G} means that any zero diagonal entry of J_K induces a zero in \hat{D} ; this corresponds to a data loss event. Any non-zero diagonal entry of J_K induces a non-zero (not identically zero) data value in \hat{D} . But the non-zero diagonal entries of J_K corresponds to non-zero columns of R and the zero diagonal entries correspond to all-zero columns of R . This proves part of the first and last statements.

Now consider a non-zero column of R . Each non-zero bit in such a column selects into an XOR formula a data or parity element from \hat{S} . Because R has zeros in row positions corresponding to zeroed positions in \hat{S} , such a formula does not depend on any lost data or parity element. The XOR formula then indicates that a specific XOR sum of known data and parity elements equals the data value associated to that column. That is, such a column provides a formula for the reconstruction. This proves the rest of the first statement in the theorem. The second claim of the theorem is clear. \square

We emphasize that this theorem makes no assumptions about the location of the failed sectors, whether they are correlated, uncorrelated or some of both. Consequently, the theorem can be applied to the case of full disk/strip losses (highly correlated) or even to the case where there is a lost sector on *every* strip (highly uncorrelated). It also does not depend on any special structure (for example geometric layout) of the erasure code. All the information we need is embedded within the generator matrix.

Recall that R is not necessarily unique and that given a basis for the null space of \hat{G} , it is easy to construct, as in (1), other pseudo-inverses that satisfy the same properties as R in the theorem. In the next section, we discuss methods for constructing pseudo-inverses and bases for null spaces. We use the null space bases for improving the sparseness of the pseudo-inverse.

6 Pseudo-inverse Constructions

There are many possible algorithms for computing pseudo-inverses and null space bases. Fundamentally, they are equivalent though the data structures and approaches differ somewhat.

From now on, we use the label B to indicate a matrix whose columns form a null space basis for some zeroed matrix \hat{G} , perhaps with all-zero column vectors as padding. Furthermore, because we are concerned only with uncorrelated sector loss, we ignore the block structure of G . As a result, we can assume without loss of generality that the generator matrix G has its systematic identity submatrix in the first N columns, with the parity columns in the right most Q columns – we call this “left systematic”. (If not, a permutation of the columns of G and corresponding column positions in \hat{G} , S , and \hat{S} and row positions of B , H and R will reduce us to this case.)

The input to our algorithms is the original generator matrix G (and/or its parity check matrix H) and a list F of data or parity elements which are declared lost (unreadable) in the stripe.

The output of our algorithms will be two matrices R and B : R is a pseudo-inverse of \hat{G} (obtained from G by zeroing the columns of G corresponding to the elements in F) and B is a basis for the null space of \hat{G} .

Our algorithms use “column operations” and/or “row operations” to manipulate matrices. Columns operations are equivalent to right multiplication by simple matrices (for rows, the operations are on the left). We consider three simplified column (or row) operations:

- Swap: exchange two columns (or rows)
- Sum and Replace: add column c to column d (modulo 2) and replace column d with the sum (similarly for rows).
- Zero: zero all the entries in a column (or row).

The first two are invertible (reversible), the Zero operation is not.

Our preferred algorithm, called the “Column-Incremental” construction, can be viewed as a dynamic or on-line algorithm. It progressively updates data structures as new lost sectors are detected (simulated by an incremental processing of the elements in F). In Section 6.3, we outline some additional constructions including static or off-line algorithms.

6.1 Column-Incremental Construction

The algorithm presented here is an incremental algorithm. It starts with a pseudo-inverse and null space basis for the matrix G (in the “good” state) and incrementally removes (simulates) a lost data or parity element, while maintaining the pseudo-inverse and null space basis properties at each step. The algorithm is space efficient and for most well-designed codes, has relatively few operations. It requires space in R only for the lost data elements (there is no need to provide recovery formulas for parity elements as these can be easily derived from the original formulas in the generator matrix – alternatively, parity columns may be added to R and so

provide additional formulas for a parity computation that reflect the lost data elements). For clarity of exposition, our description is not optimally space efficient; we leave that to the expert implementor.

The process is reversible so long as the pseudo-inverse has full rank; that is, at any step, it is possible to model reconstruction of data values for lost elements (in any order) and compute a new pseudo-inverse and null space basis equivalent to one in which the recovered elements were never lost. This is described in Section 6.4

In this algorithm, column operations are performed on a workspace matrix. The lost data or parity elements index a row of R and B .

Algorithm: Column-Incremental Construction

1. Construct a square workspace matrix W of size $(N + Q)$. In the first N columns and rows, place an identity matrix. In the last Q columns, place the parity check matrix H . Let R represent the first N columns and B represent the last Q columns of W , so $W = (R | B)$, where initially, $B = H$ and $R = \begin{pmatrix} I_N \\ 0 \end{pmatrix}$.
2. For each lost element in the list F , let r indicate the row corresponding to the lost element; perform the following operation:
 - (a) Find any column b in B that has a one in row r . If none exists, Zero any column in R that has a one in row r and continue to the next lost element. (Note that zeroing these columns zeros the entire row r in W .)
 - (b) Sum and Replace column b into every column c of W (both R and B portions) that has a one in row r .
 - (c) Zero column b in B ; equivalently, add column b to itself. Continue to the next lost element, until the list F has been processed.
3. (Optional) Use the columns of B to improve the weight of non-trivial columns of R (corresponding to lost data elements processed so far). See equation (1) and Section 6.2.
4. **Output** R (the first N columns of W) and the non-zero columns of B (from the last Q columns of W).

A proof that this algorithm satisfies the required properties can be found in the appendix of the full technical report [5]. We make the following observations.

- In practice, the workspace matrices are not very large. For example, the EVENODD code on 8 strips (with prime $p = 7$) and 16 strips (with $p = 17$) consumes only 288B and 8KB, respectively. In addition, the operations are XOR or simple pointer operations, so implementation can be very efficient. On the other hand, the invocation of this algorithm happens in an error code-path, so performance is less important than

meeting the User Contract set forth in Section 1.

- The runtime complexity of the algorithm (excluding the optimizing step 3) can be bounded by $O(|F| \cdot M^2)$ bit operations since at each of the $|F|$ steps, at most M ones can appear in row r and each such one induces M bit operations (one column operation). This is clearly an excessive upper bound as generally the matrices will be very sparse and only very few (typically $O(t)$ or $O(Q)$) ones will be in each row.

- The optimizing step 3 on R can be done either as given in a penultimate or post-processing step or during the loop after step 2c. Preferably, it is done post-processing as this step can be quite expensive (see Section 6.2). It can also be skipped; it is used to possibly minimize the XOR/IO costs but is not necessary to meet the requirements of the User Contract.

- At step 2a, there may (most likely will) be multiple choices for the column. There is no known theory that provides a criterion so that the resulting R is optimal or near optimal. One heuristic (the greedy-choice) is to use the column in B of minimal weight, but that has not always precluded a post-processing step 3 in our experiments. However, this approach does introduce the optimal formula for the current lost element (though this may change at later rounds of the loop).

An alternative heuristic is the following: in the algorithm, a column b of B is chosen with a one in position r among all such columns of B . This selected column is added to each of the others in B . This suggests that a heuristic for b is to pick the one that minimizes the total weight of the resulting columns. In 2-fault-tolerant codes, there are typically at most two such columns to choose from, so this approach is equivalent to the one of minimal weight above; this is not true for higher fault-tolerant codes.

- For only data elements (and systematic codes), it is always the case that column $c = r$ has a 1 in position r (and no other 1s elsewhere) so is always acted on in the key step. In fact, the result for this column is that we replace this column by the parity column b and then toggle the bit off in position r .

- We claim that after each lost element in the list is processed, the matrix R is a (partial or complete) pseudo-inverse for a zeroed generator matrix \hat{G} that has exactly the columns zeroed corresponding to the set of elements processed so far. This is clear in the first step because no elements have been processed, $\hat{G} = G$, the generator matrix, R is essentially an identity matrix which extracts the identity portion of G and $B = H$ is the parity check matrix, *a.k.a.* the null space basis for G . The fact that this holds true at every other step will become clear from the proof (see the appendix in the full technical report [5]).

- We never actually write down the intermediate (or

final) matrix \hat{G} . This is all handled implicitly, and so no space is used for this purpose.

- Because we perform only column operations on R , it is easy to see that what we are doing is performing, in parallel, the operations needed to determine a reconstruction formula for all lost data elements. That means that one could perform this process on individual columns as needed (e.g., to recover a single element on-demand). This would be fairly expensive globally because one repeats the same search and process algorithm on H each time, but may be marginally quicker if only one column is really needed.

- For the same reason, given the list of lost elements F , one can operate only on these columns in R and ignore all other columns. In our construction, we use all columns because in principle, we do not know what column is coming next (the algorithm does not care), so we operate on all of R at once.

- The algorithm can be used in an on-line fashion to maintain recovery formulas for lost data elements as they are detected in the stripe. As each new loss is detected, the matrices R and B get updated. If a lost element's value is reconstructed, the algorithm of Section 6.4 may be applied to again update these matrices to incorporate this new information. Alternatively, the algorithm can be applied as an off-line algorithm and applied after detection of all lost elements in the stripe.

This algorithm was a key ingredient to the results of [6] where it was applied to measure performance costs for a large variety of very different 2-fault-tolerant codes.

6.2 Improving a Pseudo-inverse

In this section we outline some approaches to implementing the optimizing step 3 in the Column-Incremental construction algorithm given above. As noted earlier, this step is not required to meet the User Contract stated in Section 1.

The following algorithm provides a systematic (though potentially very expensive) approach to finding an optimal R .

Algorithm: Improve R

1. Compute *all* the null space vectors (by taking all possible sums of subsets of the basis vectors).
2. For each non-identity (and non-zero) column of R , do the following:
 - (a) For each null space vector (from step 1), do the following:
 - i. Add the null space vector to the column of R to generate a new formula.
 - ii. If the formula generated has lower weight, then replace it in R .

3. **End**

Of course, this is only practical if the null space has small enough basis set. If the null space basis has very few vectors, then this algorithm provides an exhaustive search solution to finding an optimal R . In general, one can use any subset of the full null space to find better, but perhaps not optimal, pseudo-inverses (in Step 1 above, compute only some subset of the null space). One simple choice, is to use only the basis vectors themselves, or perhaps the basis vectors and all pairwise sums. It is an open mathematical question if there are better algorithms for finding the optimal R than that given here. However, for the extensive experiments we ran for [6], the difference between optimal and near optimal was quite minimal.

6.3 Alternative Constructions

There are alternative constructions that can be applied to computing pseudo-inverses. Among them is a Row-Incremental variation that is analogous to the Column-Incremental method described above but uses row operations instead of column operations. Most of the steps are the same as for the Column-Incremental construction. At step 2b, for each one in positions $s \neq r$ in the selected column b of B , Sum and Replace row r into row s of B ; mirror this operation in R . At step 2c zero row r in B and R and proceed to the next lost element. This algorithm has all the same properties as the column variation (including reversibility), but is typically more expensive, requiring more row operations.

Alternatively, there are both column and row versions that parallel the classical algorithm for computing an inverse. Namely, start with two matrices, the original generator matrix and an $(N + Q)$ -identity matrix. Zero the columns of the generator matrix and the identity matrix corresponding to each lost data and parity element. Perform column (or row) operations on the modified generator matrix to convert it to column (or row) reduced echelon form. Parallel each of these operations on the identity matrix; the resulting matrix contains both the pseudo-inverse and null space basis. These variations are static, off-line constructions as they utilize the complete set of lost elements in the very first step. As before, the column version has marginally less computation.

We do not give proofs for any of these constructions as they vary only slightly from the proof of the Column-Incremental construction found in the appendix of the full technical report [5]. The static algorithms can also be used to construct an initial pseudo-inverse matrix for the full generator matrix in the case when G is not systematic.

6.4 Reversing The Column Incremental Construction

As mentioned, the incremental process can be used to start with a fully on-line stripe and, step by step, as medium errors are detected in the stripe, maintain a set of reconstruction formulas (or a declaration of non-reconstructability) for every data element in the stripe. As new medium errors are detected, the matrices are updated and new formulas are generated.

It might be useful to reverse the process. Suppose the array has had some set of medium errors, but no data loss events and suppose a data element is reconstructed by its formula in R . If this reconstructed data is replaced in the stripe, it would be helpful to update the formulas to reflect this. There are two reasons for this. First, we know we can replace the formula in R by an identity column (we no longer need the old formula). But second, it may be the case that other lost elements can be reconstructed by better formulas that contain this newly reconstructed element; we should update R to reflect this fact.

One approach would be to use any algorithm to recompute from scratch the formulas for the revised set of sector losses. However, the incremental algorithm suggests that we might be able to reverse the process; that is, to update R and B directly to reflect the fact that the data element has been reconstructed (e.g., its column in R is replaced by an identity column).

To fully reverse the incremental construction of the previous section, it must be the case that no information (in the information-theoretic sense) is lost through each step. Mathematically, this happens whenever we perform a non-invertible matrix operation, i.e., that corresponds to multiplication by a non-invertible matrix. This occurs essentially in only one place in the construction: whenever we can find no vector in the null space basis with a one in the desired row. This corresponds exactly to the case where we have data loss events.

Consequently, we have the following result: so long as we never encounter the data loss branch, then (in principle), the sequence of steps can be reversed. However, the algorithm we give below works even after data loss events, so long as the restored element has a reconstruction formula in R , i.e., it is *not* itself a data loss event. Note that it makes little sense to consider restoring into the matrix an element corresponding to a data loss event (the theorem says that this is theoretically impossible).

The algorithm below performs this incremental restoration step in the case of a (recoverable) data element. Section 6.4.1 discusses the parity element case.

The input to this algorithm is a workspace matrix $W = (R | B)$ (possibly) generated by the incremental algorithm and having the property that

$$\hat{G} \cdot W = (I_N | 0)$$

where \hat{G} is the generator matrix with zeroed columns for each data or parity element in the set F of assumed lost elements (prior to a reconstruction). The other input is a data element index, that is, a row number $r \leq N$ of W . The output of the algorithm is a revised matrix W so that the above formula holds with \hat{G} having column r replaced by the appropriate identity column. The new matrix W will have an identity column in position r . (As before, the algorithm does not track the changes to \hat{G} directly, only implicitly.) Note that this process does not depend on which element is being restored from among the set of elements removed during the incremental phase (that is, it need not be the last element removed). We assume that B contains enough all-zero columns so that it has Q columns in total.

If the restored element is not from the set F , then this algorithm has no work to do, so we assume that the lost element is from F .

Algorithm: Reverse Incremental Construction

1. (Optional) For each column c in the inverse portion of W (first N columns) that has a one in every row that column r has (that is, if the AND of the columns c and r equals column r), do the following:
 - (a) Sum and Replace column r into column c ; that is, for each position of column r that has a one, set the corresponding value in column c to zero.
 - (b) Set position r in column c to the value 1.
2. Find any all-zero column b in the null space portion of W (in the last Q columns).
3. Set position (r, r) and (r, b) in W to the value 1.
4. Swap columns r and b of W .
5. (Optional) Use the null space basis vectors in B of W to reduce the weight of any column in the inverse portion R of W .
6. **Return** the updated W .

This algorithm works because it takes the reconstruction formula for the data element and unfolds it back into the null space basis, then replaces the formula with an identity column.

The first optional step replaces any occurrence of the formula for data element r in the original W by that element itself. In particular, it explicitly restores into other columns a dependence on the restored data element. In the process, it improves the weight of these formulas.

This algorithm does not necessarily completely reverse the incremental algorithm in that it does not necessarily produce identical matrices going backward as were seen going forward. However, the difference will always be something in the null space.

A proof of this construction is given in the appendix of the full technical report [5].

6.4.1 Restoring parity elements

To add a parity element back in to the matrices, we need to have the original parity column from the generator matrix G (for the data columns, we know *a priori* that this column is an identity column so we do not need to keep track of this externally). Suppose that this parity is indexed by column c in G .

Take this parity column and for each 1 in the column, sum together (modulo 2) the corresponding columns of R in W and place the result in an all-zero column of B in W . (This is exactly what we did for a data column since there was only one such column!) Replace the zero in position c of this new column by 1. Replace column c of G_0 by this parity column (restore it). (Again, this is exactly what we did for a restored data column, except we also had to set the (r, r) position in the inverse portion of W to 1 – in the case of a parity column, no such position exists in the inverse portion so this step is skipped.)

A proof is given in the appendix of the full technical report [5].

7 An Example: EVENODD Code

Consider the EVENODD(3,5) code [2] with prime $p = 3$, $n = 5$ total disks, $n - 2 = 3$ data disks and two parity disks. The data and parity layout in the strips and stripe for one instance is given in the following diagram:

S_0	S_1	S_2	P_0	P_1
$d_{0,0}$	$d_{0,1}$	$d_{0,2}$	$P_{0,0}$	$P_{0,1}$
$d_{1,0}$	$d_{1,1}$	$d_{1,2}$	$P_{1,0}$	$P_{1,1}$

The columns labeled S_0, S_1, S_2 are the data strips in the stripe (one per disk); the columns labeled P_0 and P_1 are the horizontal and diagonal parity strips, respectively. We order the data elements first by strip and then, within the strip, down the columns (this is the same view as the ordering of host logical blocks within the stripe). In this example, $N = 6$ and $Q = 4$.

The generator matrix G defined for this code is:

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

This is column blocked to indicate the strip boundaries. The matrix indicates that the parity $P_{0,1}$ is the XOR sum of the data elements indexed by the 0th, 3th, 4th and 5th rows of G , i.e.,

$$P_{0,1} = d_{0,0} + d_{1,1} + d_{0,2} + d_{1,2}. \quad (4)$$

The parity check matrix H is:

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

The parity check matrix is row blocked exactly to correspond to the column blocks of G and it contains in the lower portion an embedded identity matrix. It is easy to see that $G \cdot H = 0$; that is, H is in the null space of G (and forms a basis as well). Each column of the parity check matrix corresponds to a parity value in the array (the identity rows and the block structure provide this association).

For example, column 3 of the parity check matrix says

$$d_{0,0} + d_{1,1} + d_{0,2} + d_{1,2} + P_{0,1} = 0.$$

If this equation is not satisfied for the actual data and parity read from the disks (or detected on a channel), then an error has occurred somewhere.

More generally, we interpret these matrices in the following way. As labeled above, we consider the user data values as a row vector (ordered as already indicated):

$$D = (d_{0,0}, d_{1,0} | d_{0,1}, d_{1,1} | d_{0,2}, d_{1,2}).$$

The product $S = D \cdot G$ equals

$$(d_{0,0}, d_{1,0} | d_{0,1}, d_{1,1} | d_{0,2}, d_{1,2} | P_{0,0}, P_{1,0} | P_{0,1}, P_{1,1})$$

indicates the data layout in strips (via the block structure) as well as the formulas for computing the parity. We saw an example of this in equation (4).

The parity check matrix implies that

$$S \cdot H = 0,$$

regardless of the actual values of the data elements.

Any binary linear combination of the columns of H will also be orthogonal to all the vectors in G . E.g., take the binary sum (XOR) of columns 0 and 3 in H :

$$(1, 1 | 0, 1 | 0, 0 | 1, 0 | 0, 1)^t.$$

It is easy to see that this has the desired orthogonality property. We can replace any column in H by any such combination and still have a “parity check matrix”. Typically, the H constructed directly from the parity equations is the most sparse.

7.1 The Example – Scattered Sector Loss

Suppose we lose strip S_0 and only data element $d_{0,2}$ of S_2 in the EVENODD(3,5) code above. We then have a “zeroed” matrix \hat{G} in the form:

$$\hat{G} = \begin{pmatrix} \bullet & \bullet & & \bullet & & & \\ 0 & 0 & | 0 & 0 & | 0 & 0 & | 1 & 0 & | 1 & 0 \\ 0 & 0 & | 0 & 0 & | 0 & 0 & | 0 & 1 & | 0 & 1 \\ 0 & 0 & | 1 & 0 & | 0 & 0 & | 1 & 0 & | 0 & 1 \\ 0 & 0 & | 0 & 1 & | 0 & 0 & | 0 & 1 & | 1 & 1 \\ 0 & 0 & | 0 & 0 & | 0 & 0 & | 1 & 0 & | 1 & 1 \\ 0 & 0 & | 0 & 0 & | 0 & 1 & | 0 & 1 & | 1 & 0 \end{pmatrix}$$

where the \bullet over the column indicates the column has been removed by zeroing.

Using the data vector D , we see that we have a revised set of relationships:

$$D \cdot \hat{G} = \hat{S}, \quad (5)$$

where

$$\hat{S} = (0, 0 | d_{0,1}, d_{1,1} | 0, d_{1,2} | P_{0,0}, P_{1,0} | P_{0,1}, P_{1,1}).$$

When we view the vector \hat{S} as “known” data and parity elements (in fact, the labeled components represent the sectors that are still readable in the stripe), this equation represents a system of linear equations for the “unknown” vector D in terms of the known vector \hat{S} .

The following two matrices R and R' are easily seen to be pseudo-inverses for \hat{G} :

$$R = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \quad R' = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \quad (6)$$

We show how these matrices are obtained in Section 7.2.

The columns of R (or R') correspond to the data elements as ordered in the vector D . Each non-zero row corresponds to a position in the vector \hat{S} of known elements. Each all-zero row matches a lost element in \hat{S} . Each column represents an XOR formula for reconstructing the data element to which it corresponds. For example, to reconstruct $d_{0,2}$, we look at column 4 of R . It indicates the following formula:

$$d_{0,2} = d_{0,1} + d_{1,2} + P_{1,0} + P_{1,1},$$

and by looking at column 4 of R' we get the formula:

$$d_{0,2} = d_{1,1} + P_{0,0} + P_{1,0} + P_{0,1} + P_{1,1}.$$

It is easy to see from the original code that both of these formulas are correct (and that they do not depend on any lost sectors!).

Because the code is MDS and can tolerate two disk/strip failures, it is easy to see from dimension counting that \hat{G} has only one non-zero vector in its null space. This vector turns out to be

$$(0, 0|1, 1|0, 1|1, 0|1, 0)^t. \quad (7)$$

This is also the sum of columns 4 of R and R' (indicating that R' is derived from R by adding a vector from the null space).

The weight of each of the formulas for reconstructing data via R is at least as good as those in R' , consequently, R is a better solution than R' for our purposes. In fact, with only one vector in the null space, it is clear that R is optimal.

7.2 The Example – Constructing R

We start with the EVENODD(3,5) code as before and assume as above that data elements $d_{0,0}$, $d_{1,0}$, and $d_{0,2}$ are lost from strips S_0 and S_2 . These elements correspond to columns $r = 0, 1, 4$ of G (and also to this set of rows in our workspace).

The initial workspace is

$$W = (R | B) = \left(\begin{array}{cccccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right).$$

For row $r = 0$, we find some column in B that has a one in this row. There are two choices, $b = 6$ or $b = 8$. We choose $b = 6$ because its weight is less. We add this to columns $c = 0$ and $c = 8$ (where there is a one in row 0), then zero column $b = 6$. The result is

$$W = \left(\begin{array}{cccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right).$$

For $r = 1$, select column $b = 7$ (again, this has the minimum weight), then add this to columns $c = 1, 9$,

then zero column $b = 7$. This gives:

$$W = \left(\begin{array}{cccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array} \right).$$

Similarly, for $r = 4$ (using $b = 9$), the result is

$$W = \left(\begin{array}{cccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} \right).$$

Note that the left portion of this workspace equals R in (6). Furthermore, our null space basis B contains only the vector in (7); adding this vector to column 4 of W produces R' from (6). As R contains the optimal reconstruction formulas, no post-process step is required in this example.

It can be checked that at each stage the claimed properties of pseudo-inverse and null space of the intermediate results all hold. It should be noted that this is not against the final \hat{G} but the intermediate \hat{G} which we never write down).

7.3 The Example – Additional Sector Loss

Now suppose in addition that element $d_{0,1}$ of strip S_1 is also lost. This corresponds to a situation where sectors are lost from all three data strips of the stripe. Nominally, the EVENODD(3,5) code can only protect against losses on 2 strips; we have three partial strips, a case not covered in the literature.

The element $d_{0,1}$ corresponds to $r = 2$. We select column $b = 8$, perform the operations in the algorithm and the result is

$$W = \left(\begin{array}{cccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array} \right). \quad (8)$$

At this point, we have no more null space basis vectors (B is all zero). Any further sector loss implies a “data loss event” (see below).

Observe that any column corresponding to a data element that is *not* lost has remained unchanged as an identity column. In addition, even though we have lost sectors in three strips, all sectors are still recoverable.

If we *further* assume that data element $d_{1,1}$ (corresponding to row $r = 3$) is also lost, we can continue the algorithm. In this case, there is no null space basis vector with a one in this row. So, the algorithm says to zero all columns in R with a one in this row (that is, columns 1, 2, 3, 4). This produces the matrix

$$W = \left(\begin{array}{cccccc|cccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \right).$$

This indicates that data elements corresponding to columns 1, 2, 3, 4 are “data loss events”. However, column 0 corresponding to data element $d_{0,0}$ is still recoverable (as is $d_{1,2}$ which was never lost).

7.4 The Example – Reversing The Construction

We start with the result of our incremental construction example in equation (8) where we have lost sectors $d_{0,0}$, $d_{1,0}$, $d_{0,2}$ and $d_{0,1}$ corresponding to columns $r = 0, 1, 4, 2$ of G . Suppose we have reconstructed data element $d_{0,0}$ of column $r = 0$ (which is *not* the last element we simulated as lost). The reverse incremental algorithm above has the following steps. (We include the optional steps for completeness.)

First, we examine each of the first six columns to see if column $r = 0$ is contained in it. Column $r = 0$ has one’s in positions 5, 6, 7, 9. No other column has ones in all these positions, so we continue to the next step.

Next we select the all-zero column $b = 6$ and set position 0 in this column and in column $r = 0$ to the value 1, then we swap these two columns:

$$W = \left(\begin{array}{cccccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array} \right).$$

Next we look for null space basis elements (there’s only one to choose from) that might improve the inverse portion. For example, column 4 has weight 5. If we combine (XOR) columns 4 and 6, we get a new matrix

$$W = \left(\begin{array}{cccccc|cccc} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array} \right).$$

where the new column 4 now has weight 4. This step improved the weight of this column, as we wanted.

Note that our final result does have an identity column in position 0 so we have restored this data element.

8 Efficient Reconstruction of Partial Strips

In this section we introduce the hybrid reconstruction method. It applies the reconstruction methodology based on the matrix method in another way to address the problem of partial strip reconstruction.

Suppose the array’s erasure code can tolerate two strip failures. Most such erasure codes have a recursive algorithm defined for reconstructing the two lost strips. This can be quite efficient for rebuild of both lost strips in their entirety. The steps are generally quite simple and explicitly assume use of intermediate reconstructed elements. However, such a method will be very code-dependent; that is, the recursion will depend on the specific code layout and parity formulas. On the other hand, the matrix methodology above is completely generic. If applied without the Reverse Incremental construction, no intermediate results are used; consequently, the amount of XOR computation could be quite large compared to a recursive method. But the Reverse Incremental construction would directly take advantage of intermediate results and improve overall XOR computation costs. In fact, if applied appropriately (as a special case of our algorithm below), the matrix method (including the Reverse Incremental construction) would reduce to the recursive method in most cases (and be very similar in all others).

Now consider a host request to read a single block from one of the two lost strips (prior to completion of any background process to reconstruct the stripe). If the element is very deep into the recursion, a number of intermediate reconstructions (of lost elements) must take place; these intermediate results are not needed for the immediate host request and, though they can be cached,

are potentially extraneous work for the task at hand. The matrix method above, however, gives a (near) optimal formula for direct reconstruction of any single element without reconstruction of any additional elements.

We see that for single element reconstruction, the generic direct method of the matrix methodology is generally more efficient than the recursive method provided with a specific code. Conversely, for reconstruction of all lost elements the generally preferred method is the recursive method (either explicitly using the code's specific theory or implicitly using the matrix method together with the Reverse Incremental construction).

We now consider the problem of reconstructing a partial strip, say, to satisfy a host read for multiple consecutive blocks that span multiple elements in a strip. We assume that multiple strips are lost (though that is not a requirement at all). The above discussion suggests that neither the direct nor the recursive methods may be optimal to address this problem efficiently. We propose the following algorithm. The input to the algorithm is the set of lost sectors F , the parity check matrix (or the generator matrix) and a subset T of F containing sectors to reconstruct (we assume that no element in T is a data loss event). The output is the data values for the elements in T . That is, F is the complete set of lost sectors and T is that partial set we need to reconstruct.

Algorithm: Code-specific Hybrid Reconstruction

1. Compute the pseudo-inverse R and a (padded) null space basis for B for the lost sectors F (say, using the Column Incremental construction).
2. Do the following until all of T has been reconstructed:
 - (a) Find an unreconstructed element $t \in T$ whose reconstruction vector in R has minimal weight; reconstruct the value for t .
 - (b) Examine the recursion to see if any other element $t' \in T$ can be reconstructed by some fixed number of iterations of the recursion when starting that recursion at t . (e.g., for 2-fault-tolerant codes, this typically means at most two steps).
 - (c) If such a t' exists, reconstruct t' following the recursion; set $t \leftarrow t'$ and return to step 2b.
 - (d) If no such t' exists, do:
 - i. (Optional) Update R and B using the Reverse Incremental construction for all values reconstructed so far.
 - ii. Return to step 2a.
3. **Return** the reconstructed values for the sectors in T .

Essentially, this algorithm uses the direct method to jump into the recursion at the first point the recursion intersects the set T (thereby avoiding reconstruction of

unnneeded values). The optional step 2(d)i ensures that we have factored into the direct reconstruction formulas all values reconstructed to this point, thereby allowing these elements to be used in later reconstruction formulas (lowering XOR computational costs).

During step 2c, we can avoid physical reconstruction of intermediate steps in the recursion that are *not* in set T (that is, not immediately required for the host) by logically collapsing the recursion equations. That is, we combine the steps of the recursions to get from t to t' . This has two advantages. First, it avoids a computation and temporary memory store of any unneeded intermediate result. Second, the combination can eliminate the need for some data or parity values that appear multiply (an even number of times) in the set of recursive formulas. This avoids a possible disk read to access this data as well as the memory bandwidth costs to send this data into and out of the XOR engine multiple times.

Step 2b looks for efficient ways to utilize the recursion. If none exist, we reapply the direct method (updated, perhaps) to jump back into the recursion at some other point in T of minimal direct costs.

Together, these steps enable efficient reconstruction of only those elements that are needed (those in T) and no others. There are two special cases: (a) if T is a singleton, then this method will apply the direct method in the first step then exit; (b) if T is the union of all the elements on all lost strips, then the algorithm will default to the application of the recursion alone. We see then that this algorithm interpolates between these two extremes to find efficient reconstruction of partial strips. (Note that T need not be a partial strip, but that is the most likely application.)

More generically, we can apply the following algorithm as a means to efficiently solve the same problem, without reference to the specific recursion of the code (assuming it has one).

Algorithm: Generic Hybrid Reconstruction

1. Compute the pseudo-inverse R and a (padded) null space basis matrix B for the lost sectors F (say, using the Column Incremental Construction).
2. Do the following until all of T has been reconstructed:
 - (a) Find an unreconstructed element $t \in T$ whose reconstruction vector in R has minimal weight and reconstruct it.
 - (b) Update R and B using the Reverse Incremental construction with input t .
 - (c) Return to step 2a.
3. **Return** the reconstructed values for the sectors in T .

It is not hard to see that in the presense of a straight forward recursion, the code-specific and generic hybrid methods will produce similar results (perhaps in differ-

ent order of reconstruction, but with the same or similar costs). The application of the recursion in step 2c in the code-specific algorithm implicitly applies the Reverse Incremental algorithm.

Figure 1 shows the advantages of this hybrid method for the EVENODD code [2]. The chart shows the XOR costs (total number of XOR input and output variables) for disk array sizes from 5 to 16. These numbers are the average over all 1/2-strip-sized (element-aligned) host read requests to lost strips and averaged over all possible 2 strip failures. They are normalized to the Direct XOR costs. The figure shows that the direct cost is generally (except for very small arrays) more expensive than application of the recursive method (as one would expect for long reads), but it also shows that the Hybrid method is significantly more efficient than both.

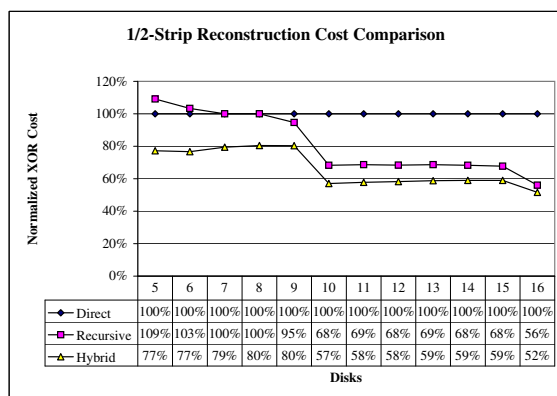


Figure 1: Comparison of Direct, Recursive and Hybrid reconstruction methods for 1/2 lost strip reconstruction, EVENODD code.

9 Summary

We developed a language to model uncorrelated and/or correlated loss of sectors (or elements) in arbitrary array codes. We provided a direct methodology and constructive algorithms to implement a universal and complete solution to the recoverability and non-recoverability of these lost sectors. This method and algorithm meets the User Contract that says that what is theoretically recoverable shall be recovered. Our solution can be applied statically or incrementally. We demonstrated the power of the direct method by showing how it can recover data in lost sectors when these sectors touch more strips in the stripe than the fault tolerance of the erasure code. The direct method can be joined with any code-specific recursive algorithm to address the problem of efficient reconstruction of partial strip data. Alternatively, the incremental method can be reversed when some data is recovered to provide a completely generic method to address this same partial strip recovery problem. Finally,

we provided numerical results that demonstrate significant performance gains for this hybrid of direct and recursive methods.

10 Acknowledgements

The authors thank Tapas Kanungo and John Fairhurst for their contributions to this work and the reviewers for helpful suggestions to improve the exposition.

References

- [1] S. Baylor, P. Corbett, and C. Park. Efficient method for providing fault tolerance against double device failures in multiple device systems, January 1999. U. S. Patent 5,862,158.
- [2] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44:192–202, 1995.
- [3] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45:46–59, 1999.
- [4] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.
- [5] V. Deenadhayalan, J. L. Hafner, KK Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. Technical Report RJ 10354, IBM Research, San Jose, CA, 2005.
- [6] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and KK Rao. Performance metrics for erasure codes in storage systems. Technical Report RJ 10321, IBM Research, San Jose, CA, 2004.
- [7] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. Northolland, Amsterdam, The Netherlands, 1977.
- [8] J. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software: Practice and Experience*, 27:995–1012, 1997.
- [9] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.
- [10] L. Xu and J. Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, IT-45:272–276, 1999.
- [11] G. V. Zaitsev, V. A. Zinovev, and N. V. Semakov. Minimum-check-density codes for correcting bytes of errors. *Problems in Information Transmission*, 19:29–37, 1983.

STAR : An Efficient Coding Scheme for Correcting Triple Storage Node Failures

Cheng Huang
Microsoft Research
One Microsoft Way,
Redmond, WA 98052

Email: cheng.huang@microsoft.com

Lihao Xu
Wayne State University
5143 Cass Avenue, 431 State Hall,
Detroit, MI 48202
Email: lihao@cs.wayne.edu

Abstract

Proper data placement schemes based on erasure correcting code are one of the most important components for a highly available data storage system. For such schemes, low decoding complexity for correcting (or recovering) storage node failures is essential for practical systems. In this paper, we describe a new coding scheme, which we call the STAR code, for correcting triple storage node failures (erasures). The STAR code is an extension of the double-erasure-correcting EVENODD code, and a modification of the generalized triple-erasure-correcting EVENODD code. The STAR code is an MDS code, and thus is optimal in terms of node failure recovery capability for a given data redundancy. We provide detailed STAR code's decoding algorithms for correcting various triple node failures. We show that the decoding complexity of the STAR code is much lower than those of the existing comparable codes, thus the STAR code is practically very meaningful for storage systems that need higher reliability.

1 Introduction

In virtually all information systems, it is essential to have a reliable data storage system that supports data availability, persistence and integrity. Here we refer to a storage system in general sense: it can be a disk array, a network of storage nodes in a clustered environment (SAN or NAS), or a wide area large scale P2P network. In fact, many research and development efforts have been made to address various issues of building reliable data storage systems to ensure data survivability, reliability, availability and integrity, including disk arrays, such as the RAID [14], clustered systems, such as the NOW [2] and the RAIN [12], distributed file systems, such as the NFS (Network File System) [39], HANFS [4], xFS [3], AFS [36], Zebra [23], CODA [37], Sprite [28], Scotch [20] and BFS [13], storage systems, such as NASD [19], Petal [25] and PASIS [42], and large

scale data distribution and archival networks, such as Intermemory [21], OceanStore [24] and Logistical Network [33].

As already indicated by these efforts, proper data redundancy is the key to provide high reliability, availability and survivability. Evolving from simple data replication or data striping in early clustered data storage systems, such as the RAID system [14], people have realized it is more economical and efficient to use the so-called *threshold schemes* to distribute data over multiple nodes in distributed storage systems [42, 41, 21, 24] than naive (multi-copy) replications. The basic idea of threshold schemes is to map an original data item into n pieces, or *shares*, using certain mathematical transforms. Then all the n shares are distributed to n nodes in the system, with each node having one share. (Each node is a storage unit, which can be a disk, a disk array or even a clustered subsystem.) Upon accessing the data, a user needs to collect at least k shares to retrieve the original data, i.e., the original data can be *exactly* recovered from m different shares if $m \geq k$, but less than k shares will not recover the original data. Such threshold schemes are called (n, k) -threshold schemes. The threshold schemes can be realized by a few means. To maximize the usage of network and storage capacity, and to eliminate bottlenecks in a distributed storage system, each data share should be of the same size. Otherwise the failure of a node storing a share with bigger size will have bigger impact on the system performance, thus creating a bottleneck in the system.

From *error control code* point of view, an (n, k) -threshold scheme with equal-size shares is equivalent to an (n, k) block code, and especially most (n, k) -threshold schemes are equivalent to (n, k) MDS (*Maximum Distance Separable*) codes [27, 26]. An (n, k) error control code uses mathematical means to transform a k -symbol message data block to an n -symbol codeword block such that any m symbols of the codeword block can recover all the k symbols of the original message

data block, where $k \leq m \leq n$. All the data symbols are of the same size in bits. Obviously by the simple *pigeon hole* principle, $k \leq m$. When $m = k$, such an (n, k) code is called MDS code, or meets the *Singleton Bound* [26]. Hereafter we simply use (n, k) to refer to any data distribution scheme using an (n, k) MDS code. Using coding terminology, each share of (n, k) is called a *data symbol*. The process of creating n data symbols from the original data whose size is of k symbols is called *encoding*, and the corresponding process of retrieving the original data from at least arbitrary k data symbols stored in the system is called *decoding*.

It is not hard to see an (n, k) scheme can tolerate up to $(n - k)$ node failures at the same time, and thus achieve data reliability, or data *survivability* in case the system is under attack where some nodes can not function normally. The (n, k) scheme can also ensure the *integrity* of data distributed in the system, since an (n, k) code can be used to detect data modifications on up to $(n - k)$ nodes. $r = n - k$ is a parameter that can describe the *reliability degree* of an (n, k) scheme.

While the concept of (n, k) has been well understood and suggested in various data storage projects, virtually all practical systems use the Reed-Solomon (RS) code [35] as an MDS code. (The so-called *information dispersal algorithm* [34] used in some schemes or systems [1] is indeed just a RS code.) The computation overhead of using the RS code, however, is large, as demonstrated in several projects, such as in OceanStore [24]. Thus practical storage systems seldom use a general (n, k) MDS code, except for full replication (which is an $(n, 1)$) or stripping without redundancy (corresponding to (n, n)) or single parity (which is $(n, n - 1)$). The advantages of using (n, k) schemes are hence very limited if not totally lost.

It is hence very important and useful to design general (n, k) codes with *both* MDS property and simple encoding and decoding operations. MDS *array codes* are such a class of codes with the both properties.

Array codes have been studied extensively [17, 22, 8, 5, 7, 43, 44, 6, 15]. A common property of these codes is that their encoding and decoding procedures use only simple binary *XOR* (*exclusive OR*) operations, which can be easily and most efficiently implemented in hardware and/or software, thus these codes are more efficient than the RS code in terms of computation complexity.

In an array code, each of the n (information or parity) symbols contain l “bits”, where a bit could be binary or from a larger alphabet. The code can be arranged in an array of size $n \times l$, where each element of the array corresponds to a bit. (When there is no ambiguity, we refer to array elements also as *symbols* for representation convenience.) Mapping to a storage system, all the symbols in a same column are stored in the same storage node.

If a storage node fails, then the corresponding column of the code is considered to be an *erasure*. (Here we adopt a commonly-used storage failure model, as discussed in [5, 15], where all the symbols are lost if the host storage node fails.)

A few class of MDS array codes have been successfully designed to recover double (simultaneous) storage node failures, i.e., in coding terminology, codes of distance 3 which can correct 2 erasures [26]. The recent ones include the EVENODD code [5] and its variations such as the RDP scheme [15], the X-Code [43], and the B-Code [44].

As storage systems expand, it becomes increasingly important to have MDS array codes of distance 4, which can correct 3 erasures, i.e., codes which can recover from *triple* (simultaneous) node failures. (There have been parallel efforts to design near-optimal codes, i.e., non-MDS codes, to tolerate triple failures, e.g. recent results from [32].) Such codes will be very desirable in large storage systems, such as the Google File System [18]. To the best of our knowledge, there exist only few classes of MDS array codes of distance 4: the generalized EVENODD code [7, 6] and later the Blaum-Roth code [9]. (There have been unsuccessful attempts resulting in codes that are not MDS [40, 29], which we will not discuss in detail in this paper.) The Blaum-Roth code is non-systematic, which requires decoding operations in any data retrieval even without node failures and thus probably is not desirable in storage systems. The generalized EVENODD code is already much more efficient than the RS code in both encoding and decoding operations. But a natural question we ask is: can its decoding complexity be further reduced? In this paper, we provide a positive answer with a new coding scheme, which we call the *STAR* code.

The STAR code is an alternative extension of the EVENODD code, a $(k + 3, k)$ MDS code which can recover triple node failures (erasures). The structure of the code is very similar to the generalized EVENODD code and their encoding complexities are also the same. Our key contribution, however, is to exploit the geometric property of the EVENODD code, and provide a new construction for an additional parity column. The difference in construction of the third parity column leads to a more efficient decoding algorithm than the generalized EVENODD code for triple erasure recovery. Our analysis shows the decoding complexity of the STAR code is very close to 3 XORs per bit (symbol), the theoretical *lower bound*, even when k is small, where the generalized EVENODD could need up to 10 XORs (Section 7) per bit (symbol). Thus the STAR code is perhaps the most efficient existing code in terms of decoding complexity when recovering from triple erasures.

It should be noted that the original generalized EVEN-

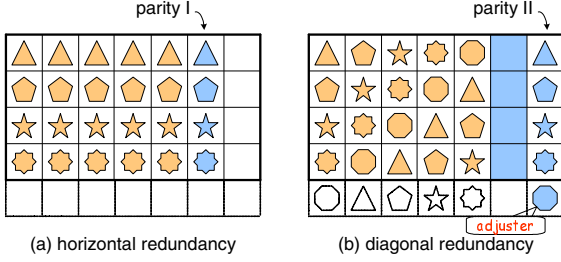


Figure 1: EVENODD Code Encoding

ODD papers [7, 6] only provide generic erasure decoding algorithms for multiple erasures. It might be possible to design a specific triple-erasure decoding algorithm to reduce decoding complexity of the generalized EVENODD. It is, however, not clear whether such a decoding algorithm for the generalized EVENODD code can achieve the same complexity as the STAR code. The interested readers thus are welcome to design an optimized triple-erasure decoding algorithm for the generalized EVENODD code and compare its performance with the STAR code.

This paper is organized as follows: we first briefly describe the EVENODD code, base on which the STAR code encoding is derived in the following section. In Section 4, we constructively prove that the STAR code can correct any triple erasures by providing detailed decoding algorithms. We also provide an algebraic description of the STAR code and show that the STAR code's distance is 4 in Section 5. We then analyze and discuss the STAR decoding complexity in Section 6 and make comparisons with two related codes in Section 7. We further share our implementation and performance tests of the STAR code in Section 8, and conclude in Section 9.

2 EVENODD Code: Double Erasure Recovery

2.1 EVENODD Code and Encoding

We first briefly describe the EVENODD code [5], which was initially proposed to address disk failures in disk array systems. Data from multiple disks form a two dimensional array, with one disk corresponding to one column of the array. A disk failure is equivalent to a column erasure. The EVENODD code uses two parity columns together with p information columns (where p is a prime number. As already observed [5, 15], p being prime in practice does not limit the k parameter in real system configuration with a simple technique called codeword shortening [26]. The code ensures that all information columns are fully recoverable when *any* two disks fail. In this sense, it is an optimal 2-erasure correcting code,

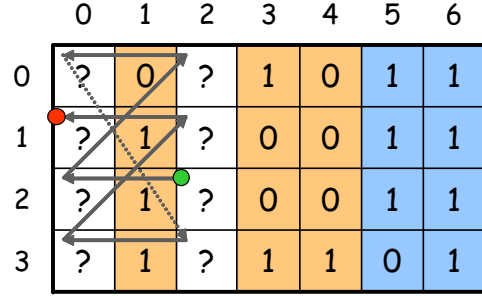


Figure 2: EVENODD Code Decoding

i.e., it is an $(p + 2, p, 3)$ MDS code. Besides this MDS property, the EVENODD code is computationally efficient in both encoding and decoding, which needs only XOR operations.

The encoding process considers a $(p - 1) \times (p + 2)$ array, where the first p columns are information columns and the last two parity columns. Symbol $a_{i,j}$ ($0 \leq i \leq p - 2, 0 \leq j \leq p + 1$) represents symbol i in column j . A parity symbol in column p is computed as the XOR sum of all information symbols in the same row. The computation of column $(p + 1)$ takes the following steps. First, the array is augmented with an imaginary row $p - 1$, where all symbols are assigned *zero* values (note that all symbols are binary ones). The XOR sum of all information symbols along the same diagonal (indeed a diagonal of *slope* 1) is computed and assigned to their corresponding parity symbol, as marked by different shapes in Figure 1. Symbol $a_{p-1,p+1}$ now becomes non-zero and is called the EVENODD *adjuster*. To remove this symbol from the array, *adjuster complement* is performed, which adds (XOR addition) the adjuster to all symbols in column $p + 1$.

The encoding can be algebraically described as follows ($0 \leq i \leq p - 2$):

$$a_{i,p} = \bigoplus_{j=0}^{p-1} a_{i,j}$$

$$a_{i,p+1} = S_1 \oplus \left(\bigoplus_{j=0}^{p-1} a_{\langle i-j \rangle_p, j} \right),$$

$$\text{where } S_1 = \bigoplus_{j=0}^{p-1} a_{\langle p-1-j \rangle_p, j}.$$

Here, S_1 is the EVENODD adjuster and $\langle x \rangle_p$ denotes $x \bmod p$. Refer to [5] for more details.

2.2 EVENODD Erasure Decoding

The EVENODD code is an optimal double erasure correcting code and any two column erasures in a coded

block can be fully recovered. Regarding to the locations of the erasures, [5] divides decoding into four cases. Here, we only summarize the most common one, where neither of the erasures is a parity column. Note that the other three cases are special ones and can be dealt with easily. A decoder first computes horizontal and diagonal *syndromes* as the XOR sum of all available symbols along those directions. Then a *starting* point of decoding can be found, which is guaranteed to be the only erasure symbol in its diagonal. The decoder recovers this symbol and then moves horizontally to recover the symbol in the other erasure column. It then moves diagonally to the next erasure symbol and horizontally again. Upon completing this *Zig-Zag* process, all erasure symbols are fully recovered. In the example shown in Figure 2 ($p = 5$), the starting point is symbol $a_{2,2}$ and the decoder moves from $a_{2,2}$ to $a_{2,0}$, $a_{0,2}$, $a_{0,0} \dots$ and finally completes at $a_{1,0}$.

3 STAR Code Encoding: Geometric Description

Extending from the EVENODD code, the STAR code consists of $p + 3$ columns, where the first p columns contain information data and the last 3 columns contain parity data. The STAR code uses the exact same encoding rules of the EVENODD code for the first two parity columns, i.e., without the third parity column, the STAR code is just the EVENODD code. The extension lies in the last parity column, column $p + 2$. This column is computed very similar to column $p + 1$, but along diagonals of slope -1 instead of slope 1 as in column $p + 1$. (The original generalized EVENODD code [7, 6] uses slope 2 for the last parity column. That is the only difference between the STAR code and the generalized EVENODD code. However, as will be seen from the following section, it is this difference that makes it much easier to design a much more efficient decoding algorithm for correcting triple erasures.) For simplicity, we call this *anti-diagonal* parity. The procedure is depicted by Figure 3, where symbol $a_{p-1,p+2}$ in parity column $p + 2$ is also an *adjuster*, similar to the EVENODD code. The adjuster is then removed from the final code block by adjuster complement. Algebraically, the encoding of parity column $p + 2$ can be represented as ($0 \leq i \leq p - 2$):

$$a_{i,p+2} = S_2 \oplus \left(\bigoplus_{j=0}^{p-1} a_{\langle i+j \rangle_p, j} \right), \text{ where } S_2 = \bigoplus_{j=0}^{p-1} a_{\langle j-1 \rangle_p, j}.$$

4 STAR Code Erasure Decoding

The essential part of the STAR code is the erasure decoding algorithm. As presented in this section, the decoding algorithm involves pure XOR operations, which allows

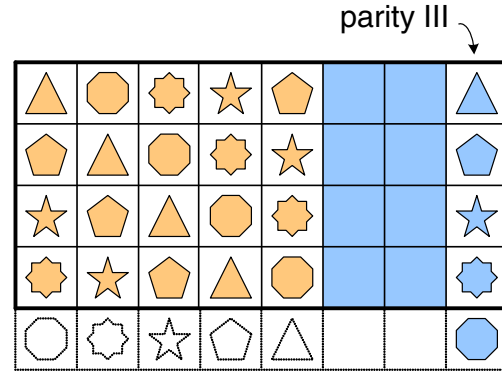


Figure 3: STAR Code Encoding

efficient implementation and thus is suitable for computation/energy constrained applications. The MDS property of the STAR code, which guarantees the recovery from arbitrary triple erasures, is explained along with the description of the decoding algorithm. A mathematical proof of this property will be given in a later section.

The STAR code decoding can be divided into two cases based on different erasure patterns: 1) decoding without parity erasures, where all erasures are information columns; and 2) decoding with parity erasures, where at least one erasure is a parity column. The former case is harder to decode and is the focus of this section. This case in turn can be divided into two subcases: symmetric and asymmetric, based on whether the erasure columns are evenly spaced. The latter case, on the other hand, handles several special situations and is much simpler.

4.1 Decoding without Parity Erasures: Asymmetric Case

We consider the recovery of triple information column erasures at position r , s and t ($0 \leq r, s, t \leq p - 1$), among the total $p + 3$ columns. (Note: hereafter, sometimes we also use r to denote a column position. It should be easy to distinguish a column position r from a code's reliability degree $r = n - k$ from the contexts.) Without loss of generality, assume $r < s < t$. Let $u = s - r$ and $v = t - s$. The asymmetric case deals with erasure patterns satisfying $u \neq v$.

The decoding algorithm can be visualized with a concrete example, where $r = 0$, $s = 1$, $t = 3$ and $p = 5$, as shown in Figure 4(a), where empty columns are erasures.

The decoding procedure consists of the following four steps:

4.1.1 Recover Adjusters and Calculate Syndromes

Given the definitions of the adjusters S_1 and S_2 , it is easy to see that they can be computed as the XOR sums of all symbols in parity columns 5, 6 and 5, 7, respectively.

Then the adjusters are assigned to symbols $a_{4,6}$, $a_{4,7}$ and also applied through XOR additions to all of the rest parity symbols in columns 6, 7, which is to reverse the adjuster complement. The redundancy property of the coded block states that the XOR sum of all symbols along any parity direction (horizontal, diagonal and anti-diagonal) should equal to *zero*. Due to erasure columns, however, the XOR sum of the rest symbols is non-zero and we denote it as the *syndrome* for this parity direction. To be specific, syndrome $\tilde{s}_{i,j}$ denotes the XOR sum of parity symbol $a_{i,j+p}$ and its corresponding non-erasure information symbols. For example, $\tilde{s}_{0,0} = a_{0,5} \oplus a_{0,2} \oplus a_{0,4}$ and $\tilde{s}_{0,1} = a_{0,6} \oplus a_{3,2} \oplus a_{1,4}$, etc. To satisfy the parity property, the XOR sum of all erasure information symbols along any redundancy direction needs to match the corresponding syndrome. For example, $\tilde{s}_{0,0} = a_{0,0} \oplus a_{0,1} \oplus a_{0,3}$ and $\tilde{s}_{0,1} = a_{0,0} \oplus a_{4,1} \oplus a_{2,3}$, etc.

In general, this step can be summarized as:

1) adjusters recovery ($j = 0, 1, 2$),

$$S_j = \bigoplus_{i=0}^{p-2} a_{i,p+j},$$

$S_1 = S_0 \oplus S_1$ and $S_2 = S_0 \oplus S_2$;

2) reversion of adjuster complement ($0 \leq i \leq p-2$),

$$\begin{aligned} a_{i,p+1} &= a_{i,p+1} \oplus S_1, \\ a_{i,p+2} &= a_{i,p+2} \oplus S_2; \end{aligned}$$

3) syndrome calculation

$$\begin{aligned} \tilde{s}_{i,0} &= a_{i,0} \oplus \left(\bigoplus_{j=0}^{p-1} a_{i,j} \right), \\ \tilde{s}_{i,1} &= a_{i,1} \oplus \left(\bigoplus_{j=0}^{p-1} a_{\langle p+i-j \rangle_p, j} \right), \\ \tilde{s}_{i,2} &= a_{i,2} \oplus \left(\bigoplus_{j=0}^{p-1} a_{\langle i+j \rangle_p, j} \right), \end{aligned}$$

where $0 \leq i \leq p-1$ and $j \neq r, s$ or t .

4.1.2 Finding a Starting Point

Recall that finding a starting point is the key step of the EVENODD decoding, which seeks one particular diagonal with only one *unknown* symbol. This symbol can then be recovered from its corresponding syndrome, and it enables the Zig-Zag decoding process until

all unknown symbols are recovered. In the STAR decoding, however, it is *impossible* to find any parity direction (horizontal, diagonal or anti-diagonal) with only one unknown symbol. Therefore, the approach adopted in the EVENODD decoding does *not* directly apply here, and additional steps are needed to find a starting point.

For illustration purpose, we now assume all syndromes are represented by the shadowed symbols in the three parity columns, as shown in Figure 4(b). Based on the diagonal parity property, it is clear that $\tilde{s}_{3,1}$ equals to the XOR sum of three unknown symbols $a_{3,0}$, $a_{2,1}$ and $a_{0,3}$, as marked by “ \triangle ” signs in Figure 4(b). Similarly, $\tilde{s}_{0,2} = a_{0,0} \oplus a_{1,1} \oplus a_{3,3}$, which are all marked by “ ∇ ” signs along an anti-diagonal. Imagine that all these marked symbols in the erasure information columns altogether form a *cross* pattern, whose XOR sum is computable ($\tilde{s}_{3,1} \oplus \tilde{s}_{0,2}$ in this case). The *key* of this step is to choose multiple crosses, such that the following two conditions are satisfied:

Condition 1

- 1) each cross is shifted vertically downward from a previous one by v symbols (offset);
- 2) the bottom row of the final cross (after wrapping around) steps over (coincides with) the top row of the first cross.

In our particular example, two crosses are chosen. The second cross is $v = 2$ symbols offset from the first one and consists of erasure symbols $a_{0,0}$, $a_{4,1}$, $a_{2,3}$ (marked by “ \triangle ”) and $a_{2,0}$, $a_{3,1}$, $a_{0,3}$ (marked by “ ∇ ”), as shown in Figure 4(c). It is straightforward that the XOR sum of these two crosses equals to $\tilde{s}_{3,1} \oplus \tilde{s}_{0,2} \oplus \tilde{s}_{0,1} \oplus \tilde{s}_{2,2}$.

Notice, on the other hand, the calculation (XOR sum) of these two crosses includes symbols $a_{0,0}$ and $a_{0,3}$ twice, the result of the bottom row of the second cross stepping over the top row of the first one. Thus, their values are canceled out and do *not* affect the result. Also notice that the parities of unknown symbol sets ($a_{2,0}$, $a_{2,1}$ and $a_{2,3}$) and ($a_{3,0}$, $a_{3,1}$ and $a_{3,3}$) can be determined by horizontal syndromes $\tilde{s}_{2,0}$ and $\tilde{s}_{3,0}$ (marked by “ \circ ”), respectively. Thus, we can get

$$a_{1,1} \oplus a_{4,1} = \tilde{s}_{3,1} \oplus \tilde{s}_{0,2} \oplus \tilde{s}_{0,1} \oplus \tilde{s}_{2,2} \oplus \tilde{s}_{2,0} \oplus \tilde{s}_{3,0},$$

as all marked in Figure 4(d).

Repeating this process and starting the first cross at different rows, we can obtain the XOR sum of any unknown symbol pair with a fixed distance 3 in column 1, i.e. $a_{0,1} \oplus a_{3,1}$, $a_{2,1} \oplus a_{0,1}$, etc.

From this example, we can see that the first condition of choosing crosses ensures the alignment of unknown symbols in the middle erasure column with those in the side erasure columns. Essentially, it groups unknown symbols together and replaces them with known syndromes. This is one way to cancel unknown symbols

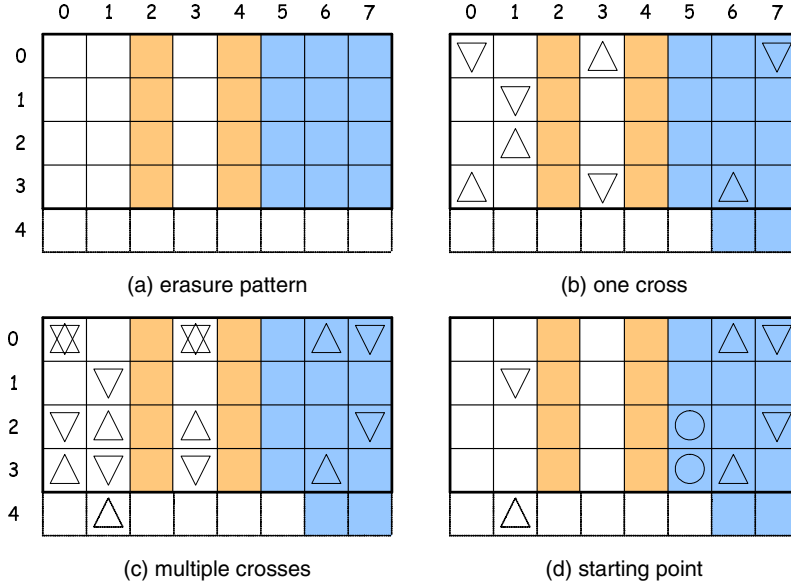


Figure 4: STAR Code Decoding

and results in a chain of crosses. The other way to cancel unknown symbols comes from the second condition, where unknown symbols in the *head* row (the first row of the first cross) of the cross chain are canceled with those in the *tail* row (the bottom row of the final cross). This is indeed “gluing” the head of the first cross with the tail of the last one and turns the chain into a *ring*. The number of crosses in the ring is completely determined by the erasure pattern (r , s and t) and the STAR code parameter p . The following Lemma 1 ensures the existence of such a ring for any given $u = s - r$, $v = t - s$ and p .

Lemma 1 *A ring satisfying Condition 1 always exists and consists of l_d ($0 \leq l_d < p$) crosses, where l_d is determined by the following equation:*

$$\langle u + l_d v \rangle_p = 0, \quad (1)$$

where $0 \leq u, v < p$.

Proof. Since p is a prime number, integers modulo p define a finite field $GF(p)$. Let v^{-1} be the unique inverse of v in this field. Then, $l_d = (p - u)v^{-1}$ exists and is unique.

Given a ring, rows with 3 unknown symbols are substituted with horizontal syndromes (*substitution*), and symbols being included even times are simply removed (*simple cancellation*). For simplicity, we refer both cases as *cancellations*. Eventually, there are exactly two rows left with unknown symbols, which is confirmed by the following Lemma 2.

Lemma 2 *After cancellations, there are exact two rows with unknown symbols in a ring. The row numbers are u and $p - u$, as offsets from the top row of the first cross.*

Proof. To simplify the proof, we only examine the ring, whose first cross starts at row 0. Now the first cross contains two unknown symbols in column r and they are in rows 0 and $u + v$. We can represent them with a polynomial $(1 + x^{u+v})$, where power values (modulo p) of x correspond to row entices. Similarly, the unknown symbols in column s can be represented as $(x^u + x^v)$. Therefore, the first cross can be completely represented by $(1 + x^{u+v} + x^u + x^v)$ and the l_1^{th} cross by

$$(1 + x^{u+v} + x^u + x^v)x^{l_1 v},$$

where $0 \leq l_1 < l_d$ and the coefficients of x are binary. Note that we don’t explicitly consider unknown symbols in column t , which are reflected by polynomials representing column r . Using this representation, the cancellation of a polynomial term includes both cases of substitution and simple cancellation. The XOR sum of all crosses is as

$$\begin{aligned} & \sum_{l_1=0}^{l_d-1} (1 + x^{u+v} + x^u + x^v)x^{l_1 v} \\ &= (1 + x^u) \sum_{l_1=0}^{l_d-1} (1 + x^v)x^{l_1 v} \\ &= (1 + x^u)(1 + x^{p-u}) \\ &= x^u + x^{p-u}, \end{aligned} \quad (2)$$

where l_d is substituted using the result from Lemma 1. Thus, only two rows with unknown symbols are left after cancellations and the distance between them is $d = \langle p - 2u \rangle_p$.

It is important to point out that unknown symbols in the remaining two rows are *not* necessarily in column s . For example, if $r = 0$, $s = 2$ and $t = 3$, the remaining unknown symbols would be $a_{2,0}$, $a_{2,3}$, $a_{3,0}$ and $a_{3,3}$, which are indeed columns r and t . However, it is conceivable that we can easily get the XOR sum of corresponding unknown symbol pair in column s , since horizontal syndromes are available.

To summarize this step, we denote l_h to be the number of rows in a ring, which are canceled through substitution and define the set of corresponding row indices as $F_h = \{h_{l_2} \mid 0 \leq l_2 < l_h\}$. The set F_h is simply obtained by enumerating all crosses of the ring and then counting rows with 3 unknown symbols. Let \tilde{a}_u denote the XOR sum of the unknown symbol pair $a_{0,s}$ and $a_{\langle p-2u \rangle_p, s}$, then the i^{th} pair has

$$\tilde{a}_{u+i} = \bigoplus_{l_1=0}^{l_d-1} \tilde{s}_{\langle -r+i \rangle_p, 2} \bigoplus_{l_2=0}^{l_h-1} \tilde{s}_{\langle h_{l_2}+i \rangle_p, 0} \bigoplus_{l_1=0}^{l_d-1} \tilde{s}_{\langle t+i \rangle_p, 1} \quad (3)$$

where $0 \leq i \leq p-1$.

4.1.3 Recover Middle Erasure Column

In the previous step, we have computed the XOR sum of arbitrary unknown symbol pair in column s with the fixed distance 3. Since symbol $a_{4,1}$ is an imaginary symbol with zero value, it is straightforward to recover symbol $a_{1,1}$. Next, symbol $a_{3,1}$ can be recovered since the XOR sum of the pair $a_{1,1}$ and $a_{3,1}$ is available. Consequently, symbols $a_{0,1}$ and $a_{2,1}$ are recovered. This process is shown to succeed with arbitrary parameters by Lemma 3.

Lemma 3 *Given the XOR sum of arbitrary symbol pair with a fixed distance d , all symbols in the column are recoverable if there is at least one symbol available.*

Proof. Since p is prime, $F = \{\langle di \rangle_p \mid 0 \leq i \leq p-1\}$ covers all integers in $[0, p)$. Therefore, a “tour” starting from row $p-1$ with the stride size d will visit all other rows exactly once before returning to it. As the symbol in row $p-1$ is always available (zero indeed) and the XOR sum of any pair with distance d is also known, all symbols can then be recovered along the tour.

To summarize, this step computes

$$\tilde{a}_{\langle (p-1)-di \rangle_p} = \tilde{a}_{\langle (p-1)-di \rangle_p} \oplus a_{\langle (p-1)-d(i-1) \rangle_p}, \quad (4)$$

where $0 \leq i \leq p-1$. Then, $a_{i,s} = \tilde{a}_i$ (where there are 2 unknown symbols left in the ring after cancellations) or

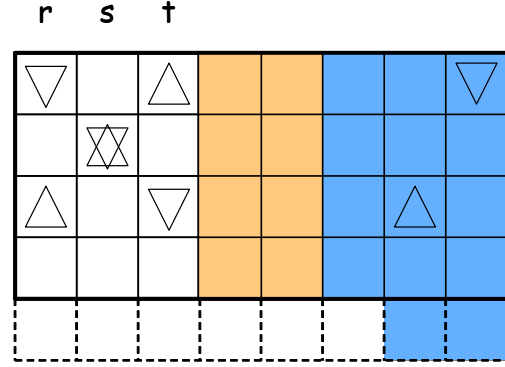


Figure 5: STAR Code Decoding (Symmetric Erasures)

$a_{i,s} = \tilde{a}_i \oplus \tilde{s}_{i,0}$ (where 4 unknown symbols are left) for all i 's. Thus far, column s is completely recovered.

4.1.4 Recover Side Erasure Columns

Now that column s is known, the first $p+2$ columns compose an EVENODD coded block with 2 erasures. Thus this reduces to an EVENODD decoding of two erasures.

4.2 Decoding without Parity Erasures: Symmetric Case

When the erasure pattern is symmetric ($u = v$), the decoding becomes much easier, where step 2 is greatly simplified while all other steps remain the same.

To illustrate the step of finding a starting point, we still resort to the previous example, although the erasure pattern is different now. Let's assume $r = 0$, $s = 1$ and $t = 2$, as shown in Figure 5. It is easy to see that only one cross is needed to construct a “ring” (still denoted as a ring, although not closed anymore). As in this example, a cross consists of unknown symbols $a_{0,0}$, $a_{0,2}$, $a_{2,0}$ and $a_{2,2}$, and $a_{1,1}$ is canceled because it is included twice. The XOR sum of the cross thus equals to $\tilde{s}_{2,1} \oplus \tilde{s}_{0,2}$. This is very similar to the situation in the previous case, where there are 4 unknown symbols in a ring after cancellations. Therefore, the rest of the decoding can followed the already described procedure and we don't repeat in here.

In summary the symmetric case can be decoded using the procedure for the asymmetric case, by simply setting $l_d = 1$, $l_h = 0$, $u = 0$ and $d = t - r$.

4.3 Decoding with Parity Erasures

In this part, we consider the situation when there are erasures in parity columns. The decoding is divided into the following 3 subcases.

4.3.1 Column $p + 2$ is an erasure

This then reduces to EVENODD decoding of two erasures. Note that this case also takes care of all patterns with fewer than 3 erasures.

4.3.2 Column $p + 1$ is an erasure, while $p + 2$ is not

This is almost the same as the previous case, except that now the “EVENODD” coded block consists of the first $p + 1$ columns and column $p + 2$. In fact, this coded block is no longer a normal EVENODD code, but rather a mirror reflection of one over the horizontal axis. Nevertheless, it can be decoded with slightly modification of the EVENODD decoding, which we simply leave to interested readers.

4.3.3 Column p is an erasure, while $p + 1$ and $p + 2$ are not

In this case, $0 \leq r < s \leq p - 1$ and $t = p$.

First, it is not possible to recover adjusters S_1 and S_2 , as symbols in column p are unknown. However, $S_1 \oplus S_2$ is still computable, which simply equals to the XOR sum of all symbols in column $p + 1$ and $p + 2$. This is easy to see from the definitions of S_1 and S_2 , S_0 is added twice and canceled out. It is thus possible to reverse the adjuster complement. The results from syndrome calculation are XOR sums of syndromes and their corresponding adjusters, rather than syndromes themselves. We use $\hat{s}_{i,j}$ to denote the results, which thus satisfy

$$\hat{s}_{i,j} = \tilde{s}_{i,j} \oplus S_j, \quad (5)$$

where $j = 1$ or 2 and $0 \leq i \leq p - 1$. Note that $\hat{s}_{i,0} = \tilde{s}_{i,0}$ for all i 's.

The next step is similar to the decoding of the symmetric case without parity erasures, as it is also true that only one cross is needed to construct a ring. Taking the cross starting with row 0 as an example, it consists of unknown symbols $a_{0,r}$, $a_{0,s}$, $a_{u,r}$ and $a_{u,s}$. Since the XOR sum of this cross equals to $\tilde{s}_{s,1} \oplus \tilde{s}_{(-r)p,2}$, we can easily get the following equation by substituting Eq. 5:

$$a_{0,r} \oplus a_{0,s} \oplus a_{u,r} \oplus a_{u,s} = \hat{s}_{s,1} \oplus \hat{s}_{(-r)p,2} \oplus S_1 \oplus S_2.$$

Therefore, the XOR sum of the cross is computable. Following the approach as used to recover middle erasure column in an early section, the XOR sum of two unknown symbols on any row can be recovered, which is still denoted as \tilde{a}_i ($0 \leq i \leq p - 1$). Then, parity column p can be recovered, as

$$a_{i,p} = \tilde{a}_i \oplus \tilde{s}_{i,0} = \tilde{a}_i \oplus \hat{s}_{i,0},$$

where $0 \leq i \leq p - 1$.

After column p is recovered, the first $p + 2$ columns can again be regarded as an EVENODD coded block with 2 erasures at column r and s . Therefore, the application of the EVENODD decoding can complete the recovery of all the remaining unknown symbols.

To summarize the procedure in this subcase, we have

$$S_1 \oplus S_2 = \left(\bigoplus_{i=0}^{p-2} a_{i,p+1} \right) \oplus \left(\bigoplus_{i=0}^{p-2} a_{i,p+2} \right),$$

and

$$\begin{aligned} \hat{s}_{i,0} &= a_{i,0} \oplus \left(\bigoplus_{j=0}^{p-1} a_{i,j} \right), \\ \hat{s}_{i,1} &= a_{i,1} \oplus \left(\bigoplus_{j=0}^{p-1} a_{\langle p+i-j \rangle_p, j} \right), \\ \hat{s}_{i,2} &= a_{i,2} \oplus \left(\bigoplus_{j=0}^{p-1} a_{\langle i+j \rangle_p, j} \right), \end{aligned}$$

where $0 \leq i \leq p - 1$ and $j \neq r$ or s . Then,

$$\tilde{a}_i = \hat{s}_{\langle s+i \rangle_p, 1} \oplus \hat{s}_{\langle -r+i \rangle_p, 2} \oplus S_1 \oplus S_2,$$

where $0 \leq i \leq p - 1$, and

$$\tilde{a}_{\langle (p-1)-ui \rangle_p} = \tilde{a}_{\langle (p-1)-ui \rangle_p} \oplus a_{\langle (p-1)-u(i-1) \rangle_p},$$

where $1 \leq i \leq p - 1$. Finally, column p can be recovered as

$$a_{i,p} = \tilde{a}_i \oplus \hat{s}_{i,0},$$

for all i 's. The rest is to use the EVENODD decoding to recover the remaining 2 columns, which is skipped in here.

Putting all the above cases together, we conclude this section with the following theorem:

Theorem 1 *The STAR code can correct any triple column erasures and thus it is a $(p + 3, p)$ MDS code.*

5 Algebraic Representation of the STAR Code

As described in [5], each column in the EVENODD code can be regarded algebraically as an element of a polynomial ring, which is defined with multiplication taken modulo $M_p(x) = (x^p - 1)/(x - 1) = 1 + x + \dots + x^{p-2} + x^{p-1}$. For the ring element x , it is shown that its multiplicative order is p . Using β to denote this element, then column j ($0 \leq j \leq p + 1$) can be represented using the notation $a_j(\beta) = a_{p-2,j}\beta^{p-2} + \dots + a_{1,j}\beta + a_{0,j}$, where $a_{i,j}$ ($0 \leq i \leq p - 2$) is the i^{th} symbol in the column. Note that the multiplicative inverse of β exists and

can be denoted as β^{-1} . Applying same notations to the STAR code, we can then get its parity check matrix as:

$$H = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 & 0 & 0 \\ 1 & \beta & \cdots & \beta^{p-1} & 0 & 1 & 0 \\ 1 & \beta^{-1} & \cdots & \beta^{-(p-1)} & 0 & 0 & 1 \end{bmatrix} \quad (6)$$

It is not hard to verify that, as in [7], that any 3 columns in the parity check matrix are linearly independent. Therefore, the minimum distance of the STAR code is indeed 4 (each column is regarded as a single element in the ring) and thus arbitrary triple (column) erasures are recoverable. This is an alternative way to show its MDS property.

6 Complexity Analysis

In this section, we analyze the complexity of the STAR code erasure decoding. The complexity is dominated by XOR operations, thus we can count the total number of XORs and use that as an indication of the complexity. Since decoding without parity erasures is the most complicated case, including both asymmetric and symmetric erasure patterns, our analysis is focused on this case.

6.1 Erasure Decoding Complexity

It is not difficult to see that the complexity can be analyzed individually for each of the 4 decoding steps. Note that a complete STAR code consists of p information columns and $r = n - k = 3$ parity columns. When there are only k ($k \leq p$) information columns, we can still use the same code by resorting to the *shortening* technique, which simply assigns zero value to all symbols in the last $p - k$ information columns. Therefore, in the analysis here, we assume the code block is a $(p - 1) \times (k + 3)$ array.

In step 1, the calculation of S_0 takes $(p - 2)$ XOR operations and those of S_1 and S_2 take $(p - 1)$ XORs each. The reversion of adjuster complement takes $2(p - 1)$ XORs in total. Directly counting XORs of the syndrome calculations is fairly complicated and we can resort to the following alternative approach. First, it is easy to see that the syndrome calculations of any parity direction for a code block without erasures (a $(p - 1) \times (p + 3)$ array) take $(p - 1)p$ XORs. Then, notice that any information column contributes $(p - 1)$ XORs to the calculations. Therefore, for a code block with $(k - 3)$ information columns (with triple erasures), the number of XORs becomes $(p - 1)p - (p - k + 3)(p - 1) = (k - 3)(p - 1)$. In total, the XORs in this step is:

$$\begin{aligned} & (p - 2) + 2(p - 1) + 2(p - 1) + 3(k - 3)(p - 1) \\ &= (3k - 4)(p - 1) - 1. \end{aligned} \quad (7)$$

In step 2, the computation of each ring takes $(2l_d + l_h - 1)$ XORs and there are $(p - 1)$ rings to compute. Thus, the number of XORs is

$$(2l_d + l_h - 1)(p - 1). \quad (8)$$

In step 3, it is easy to see that the number of XORs is

$$(p - 1) - 1 = p - 2. \quad (9)$$

In step 4, the horizontal and the diagonal syndromes need to be updated with the recovered symbols of column s , which takes $2(p - 1)$ XORs. Note that there is no need to update the anti-diagonal syndromes, because the decoding hereafter deals with only double erasures. The Zig-Zag decoding then takes $2(p - 1) - 1$ XORs. So the number of XORs in this step is

$$2(p - 1) + 2(p - 1) - 1 = 4(p - 1) - 1. \quad (10)$$

Note that in step 2, the number of XORs is computed assuming the case where only 2 unknown symbols are left in a ring after cancellations. If the other case happens, where 4 unknown symbols are left, additional $(p - 1)$ XOR operations are needed to recover column s . However, this case does *not* need to update the horizontal syndromes in step 4 and thus saves $(p - 1)$ XORs there. Therefore, it is just a matter of moving XOR operations from step 2 to step 4 and the total number remains the same for both cases.

In summary, the total number of XORs required to decode triple information column erasures can be obtained by putting Eq. (7), (8), (9) and (10) together, as:

$$\begin{aligned} & (3k - 4)(p - 1) - 1 + (2l_d + l_h - 1)(p - 1) \\ &+ (p - 2) + 4(p - 1) - 1 \\ &= (3k + 2l_d + l_h)(p - 1) - 3 \end{aligned} \quad (11)$$

$$\approx (3k + 2l_d + l_h)(p - 1). \quad (12)$$

6.2 A Decoding Optimization

From Eq. (12), we can see that for fixed code parameters k and p , the decoding complexity depends on l_d and l_h , which are completely determined by actual erasure patterns (r , s and t). In Sec. 4, we present an algorithm to construct a ring of crosses, which will yield a starting point for successful decoding. Within the ring, all crosses are $v = t - s$ symbols offset from previous ones. From Eq. (2), there are exactly two rows with unknown symbols left after cancellations. From the symmetric property of the ring construction, it is not difficult to show that using offset $u = s - r$ will also achieve the same goal. If using u as offset results in smaller l_d and l_h values (to be specific, smaller $2l_d + l_h$), then there is advantage to do so.

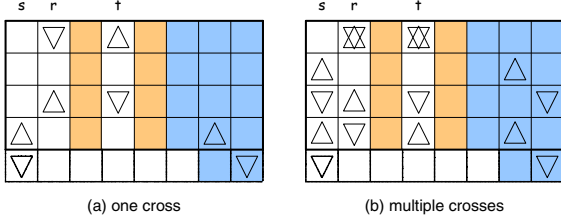


Figure 6: Optimization of STAR Decoding

Moreover, we make the assumption $r < s < t$ during the description of the decoding algorithm. Although it helps to visualize the key procedure of finding a starting point, this assumption is unnecessary. Indeed, it is easy to verify that all proofs in Sec. 4 still hold without this assumption. By swapping values among r , s and t , it might be possible to reduce the decoding complexity. For instance, in the previous example, $r = 0$, $s = 1$ and $t = 3$ results in $l_d = 2$ and $l_h = 2$. If letting $r = 1$, $s = 0$ and $t = 3$, then $u = -1$ and $v = 3$. The pattern of single cross is shown in Figure 6(a). From Figure 6(b), it is clear that two crosses close a ring, which contains exactly two rows (row 1 and 4) with unknown symbols after cancellations. Thus, this choice also yields $l_d = 2$ and $l_h = 2$. However, if letting $r = 0$, $s = 3$ and $t = 1$, we can get $u = s - r = 3$ and $v = t - s = -2$. It is easy to find out that unknown symbols in column s are canceled in every single cross. In fact, this is an equivalence of the symmetric case and in turn $l_d = 1$ and $l_h = 0$. Thus, the complexity is reduced by this choice. Note that for general u and v , the condition of symmetric now becomes $\langle u - v \rangle_p = 0$, instead of simply $u = v$.

Now let us revisit the ring construction algorithm described in Sec. 4. The key point there is to select multiple crosses such that the bottom row of the final cross “steps over” the top row of the first one, and there are exact two rows left with unknown symbols after cancellations. Further examination, however, reveals that it is possible to construct rings using alternative approaches. For instance, the crosses can be selected in such a way that *in the middle column* the bottom symbol of the final cross “steps over” the top symbol of the first one. Or perhaps there is even no need to construct closed rings and crosses might not have to be a fixed offset from previous ones. Indeed, if crosses can be selected arbitrarily while still ensuring exact two rows left with unknown symbols after cancellations, the successful decoding can be guaranteed. Recall that single cross is represented by $C(x) = 1 + x^u + x^v + x^{u+v}$ and a cross of f symbol offset by $C(x)x^f$. Therefore, the construction of a ring is to determine a polynomial term $R(x)$, such that $C(x)R(x)$ results in exact two entries. For instance, the example in Sec. 4 has $R(x) = 1 + x^2$ and $C(x)R(x) = x + x^4$. It

is thus possible to further reduce the decoding complexity. Theorem 2 shows that the decoding complexity is minimized if a $R(x)$ with minimum entries is adopted.

Theorem 2 *The decoding complexity is nondecreasing with respect to the number of crosses (l_d) in a ring.*

Proof. Whenever a new cross is included into the ring, two new non-horizontal syndromes (one diagonal and one anti-diagonal) need to be added to the XOR sum. With this new cross, at most four rows can be canceled (simple cancellation due to even times addition), among which two can be mapped with this cross and the other two with an earlier cross. Thus, each cross adds two non-horizontal syndromes but subtracts at most two horizontal syndromes. The complexity is thus nondecreasing with respect to the number of crosses.

Note that l_d is in fact the number of entries in $R(x)$. An optimal ring needs to find a $R(x)$ with minimum entries, which then ensures that $C(x)R(x)$ has only two terms. An efficient approach to achieve this is to test all polynomials with two terms. If a polynomial is divisible by $C(x)$, then the quotient yields a valid $R(x)$. A $R(x)$ with minimum entries is then chosen to construct the ring. It is important to point out that there is no need to worry about common factors (always powers of x) between two terms in the polynomial, as it is not divisible by $C(x)$. Thus, the first entry of all polynomials can be fixed as 1, which means that only $p - 1$ polynomials ($1 + x^i$, $0 < i \leq p - 1$) need to be examined. As stated in an earlier section, polynomials are essentially elements in the ring constructed with $M_p(x) = 1 + x + \dots + x^{p-2} + x^{p-1}$. Based on the argument in [8], $(1 + x^u)$ and $(1 + x^v)$ are invertible in the ring. Thus, $C(x) = (1 + x^u)(1 + x^v)$ is also invertible, and it is straightforward to compute the inverse using Euclid’s algorithm. For instance, $C(x) = 1 + x + x^2 + x^3$, as $u = 1$ and $v = 2$ in the previous example. The generator polynomial $M_p(x) = 1 + x + x^2 + x^3 + x^4$ as $p = 5$. Applying the Euclid’s algorithm [26], it is clear that

$$1(1 + x + x^2 + x^3 + x^4) + x(1 + x + x^2 + x^3) = 1. \quad (13)$$

Thus, the inverse of $C(x)$ is $inv(C(x)) = x$. When examining the polynomial $1 + x^3$, we get $R(x) = inv(C(x))(1 + x^3) = x + x^4$ or equivalently,

$$(1 + x + x^2 + x^3)(x + x^4) = 1 + x^3 \text{ mod } M_p(x). \quad (14)$$

It is desirable that $R(x)$ carries the entry of power 0, since the ring always contains the original cross. So we multiply x to both sides of Eq. (14), which now becomes

$$(1 + x + x^2 + x^3)(1 + x^2) = x + x^4 \text{ mod } M_p(x).$$

Thus, we have $R(x) = 1 + x^2$ and the ring can be constructed using two crosses ($l_d = 2$) with an offset of two

symbols. Once the ring is constructed, it is straightforward to get l_h .

Note this optimal ring construction only needs to be computed once in advance (offline). Thus we do not count the ring construction in the decoding procedure.

7 Comparison with Existing Schemes

In this section, we compare the erasure decoding complexity of the STAR code to two other XOR-based codes, one proposed by Blaum *et al.* [7] (Blaum code hereafter) and the other by Blomer *et al.* [10].

The Blaum code is a generalization of the EVENODD code, whose horizontal (the 1^{st}) and diagonal (the 2^{nd}) parities are now regarded as redundancies of slope 0 and 1, respectively. A redundancy of slope $q-1$ ($q \geq 3$) generates the q^{th} parity column. This construction is shown to maintain the MDS property for triple parity columns, when the code parameter p is a prime number. The MDS property continues to hold for selected p values when the number of parities exceeds 3. To make the comparison meaningful, we focus on the triple parity case of the Blaum code. We compare the complexity of triple erasure decoding in terms of XOR operations between the Blaum code and the STAR code. As in the previous sections, we confine all three erasures to information columns.

The erasure decoding of the Blaum code adopts an algorithm described in [8], which provides a general technique to solve a set of linear equations in a polynomial ring. Due to special properties of the code, however, ring operations are *not* required during the decoding procedure, which can be performed with pure XOR and shift operations. The algorithm consists of 4 steps, whose complexities are summarized as follows: 1) syndrome calculation: $3(k-3)(p-1)-1$; 2) computation of $\hat{Q}(x; z)$: $\frac{1}{2}r(3r-3)p$; 3) computation of the right-hand value: $r((r-1)p + (p-1))$; and 4) extracting the erasure values: $r(r-1)(2(p-1))$. Here $r = n - k = 3$. Therefore, the total number of XORs is

$$\begin{aligned} & 3(k-3)(p-1) - 1 + 9p + (9p-3) + 12(p-1) \\ &= (3k+21)(p-1) + 14 \end{aligned} \quad (15)$$

$$\approx (3k+21)(p-1). \quad (16)$$

Comparison results with the STAR code are shown in Figure 7, where we can see that the complexity of the STAR decoding remains fairly constant and is just slightly above 3. Note that this complexity depends on actual erasure locations, thus the results reported here are average values over all possible erasure patterns. The complexity of the Blaum code, however, is rather high for small k values, although it *does* approach 3 asymptotically. The STAR code is thus probably more desirable than the Blaum code. Figure 7 also includes the

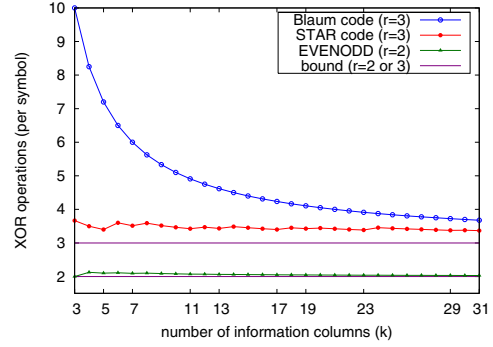


Figure 7: The Complexity Comparisons ($r = n - k$)

complexity of the EVENODD decoding as a reference, which is roughly constant and slightly above 2 XORs per symbol. Note in Figure 7, p is always taken for each k as the next largest prime.

Further reflection on the Blaum code and the STAR code would reveal that the construction difference between them lies solely on the choice of the 3^{rd} redundancy slope, where the Blaum code uses slope 2 and the STAR code -1 . One might wonder whether the decoding approach adopted here could be applied to the Blaum code as well. Based on STAR decoding's *heavy* reliance on the geometric property of individual crosses in the step to find a starting point, it seems difficult to achieve the same ring construction in the Blaum code when symmetry is no longer obvious. Moreover, the intuitiveness of the decoding process would be completely lost even if it is possible at all. Instead, we would be more interested to investigate whether the STAR code construction, so as the decoding approach, could be extended to handle more than triple erasures, as the Blaum code already does.

The XOR-based code proposed in [10] uses Cauchy matrices to construct a Reed-Solomon (RS) code. It replaces generator matrix entries, information and parity symbols with binary representations. Then, the encoding and decoding can be performed with primarily XOR operations. To achieve maximum efficiency, it requires message length to be multiples of 32 bits. In that way, basic XOR unit is 32 bits, or single word, and can be performed by single operation. To compare with this scheme fairly, we require the symbol size of the STAR code to be multiples of 32 bits too. It is shown that the XOR-based decoding algorithm in [10] involves krL^2 XOR operations and r^2 operations in a finite field $GF(2^L)$, where k and r are the numbers of information symbols and erasures, respectively. We ignore those r^2 finite field operations (due to the inversion of a decoding coefficient matrix), which tend to be small as the number of erasures is limited. Then, the RS code's non-

# of total columns (n)	# of XORs (= rL)	
	$r = 2$	$r = 3$
$n \leq 8$	6	9
$9 < n \leq 16$	8	12
$17 < n \leq 32$	10	15
$33 < n \leq 64$	12	18

Table 1: Complexity of the RS Code (per 32 bits)

malized decoding complexity (by the total information length of kL words) is rL . As the total number of symbols $n (= k + r)$ is limited by L ($n \leq 2^L$), we have to increase L and thus in turn the decoding complexity when n increases (see Table 1). Compared to Figure 7, where the STAR code decoding complexity is slightly more than 3 XORs per symbol (multiples of 32 bits now), it is clear that the STAR code is much more efficient than the XOR-based RS code. Note that the complexity of *normal* (finite field-based) RS code implementation (e.g. [30]) turns out to be even higher than the XOR-based one, so we simply skip comparison here.

8 Implementation and Performance

The implementation of the STAR code encoding is straightforward, which simply follows the procedure described in Sec. 3. Thus, in this part, our main focus is on the erasure decoding procedure. As stated in Sec. 6, the decoding complexity is solely determined by l_d and l_h , given the number of information columns k and the code parameter p . As l_d and l_h vary according to actual erasure patterns, so does the decoding complexity. To achieve the maximum efficiency, we apply the optimization technique as described in the earlier section.

An erasure pattern is completely determined by the erasure columns r , s and t (again assume $r < s < t$), or further by the distances u and v between these columns, as the actual position of r does *not* affect l_d or l_h . Therefore, it is possible to set up a mapping from (u, v) to (l_d, l_h) . To be specific, given u and v , the mapping returns the positions of horizontal, diagonal and anti-diagonal syndromes, which would otherwise be obtained via ring constructions. The mapping can be implemented as a lookup table and the syndrome positions using bit vectors. Since the lookup table can be built in advance of actual decoding procedure, it essentially shifts complexity from online decoding to offline preprocess. Note that the table lookup operation is only needed once for every erasure pattern, thus there is no need to keep the table in memory (or cache). This is different from finite field based coding procedures, where intensive table lookups are used to replace complicated finite field operations. For example, a RS code implementation might use an

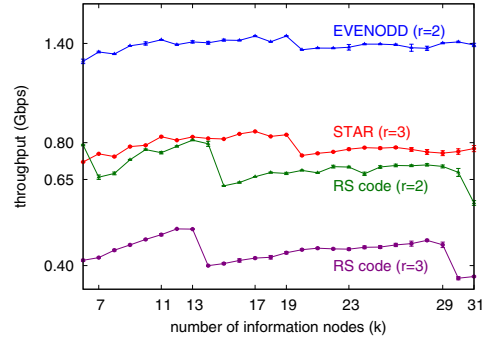


Figure 8: Throughput Performance. ($r = n - k$ erasures are randomly generated among information nodes.)

exponential table and a logarithm table for *each* multiplication/division. Furthermore, the number of entries in the lookup table is not large at all. For example, for code parameter $p = 31$, u and v are at most 30, which requires a table of at most $30 \times 30 = 900$ entries, where each entry contains 3 bit vectors (32-bit each) for the ring construction, one byte for the decoding pattern and another byte for l_h . The cost of maintaining a few tables of this size is then negligible.

During the decoding procedure, u and v are calculated from the actual erasure pattern. Based on these values, the lookup table returns all syndrome positions, which essentially indicates the ring construction. The calculation of the ring is thus performed as the XOR sums of all the indicated syndromes. Then, the next ring is calculated by offsetting all syndromes with one symbol and the procedure continues until all rings are computed. Steps afterward are to recover the middle column and then the side columns, as detailed in Sec. 4.

We implement the STAR code erasure decoding procedure and apply to reliable storage systems. The throughput performance is measured and compared to the publicly available implementation of the XOR-based RS code [11]. The results are shown in Figure 8, where the size of a single data block from each node is 2880 bytes and the number of information storage nodes (k) varies from 6 to 31. Note our focus is on decoding erasures that all occur at information columns, since otherwise the STAR code just reduces to the EVENODD code (when there is one parity column erasure) or a single parity code (when there are two parity column erasures), so we only simulate random information column erasures in Figure 8. Recall that a single data block from each node corresponds to a single column in the STAR code and is divided into $p - 1$ symbols, so the block size needs to be a multiple of $p - 1$. For comparison purpose, we use 2880 here since it is a common multiple of $p - 1$ for most p values in the range. In real applications, we are free to

choose the block size to be any multiple of $p - 1$, once p , as a system parameter, is determined. These results are obtained from experiments on a P3 450MHz Linux machine with 128M memory running Redhat 7.1. It is clear that the STAR code achieves about twice throughput compared to the RS code. Note that there are jigsaw effects in the throughputs of both the EVENODD and the STAR code. This happens mainly due to the shortening technique. When the number of storage nodes is not prime, the codes are constructed using the closest larger prime number. A larger prime number means each column (data block here) is divided into more pieces, which in turn incurs additional control overhead. As the number of information nodes increases, the overhead is then amortized, reflected by the performance ramping up after each dip. (Similarly, the performance of the RS code shows jigsaw effects too, which happens at the change of L due to the increment of total storage nodes n .) Moreover, note that the throughputs are not directly comparable between $r (= n - k) = 2$ and $r (= n - k) = 3$ (e.g. the EVENODD and the STAR code), as they correspond to different reliability degrees. The results of codes with $r = 2$ are depicted only for reference purpose. Finally, note that necessary correction of the generator matrix (similar to the one documented in [31]) needs to be done in the aforementioned implementation of the XOR-based RS code to ensure the MDS property. This doesn't affect the throughput performance though.

9 Conclusions

In this paper, we describe the STAR code, a new coding scheme that can correct triple erasures. The STAR code extends from the EVENODD code, and requires only XOR operations in its encoding and decoding operations. We prove that the STAR code is an MDS code of distance 4, and thus is optimal in terms of erasure correction capability vs. data redundancy. Detailed analysis shows the STAR code has the lowest decoding complexity among the existing comparable codes. We hence believe the STAR code is very suitable for achieving high availability in practical data storage systems.

Acknowledgments

The authors wish to thank anonymous reviewers for their very insightful and valuable comments and suggestions, which certainly help improve the quality of this paper. This work was in part supported by NSF grants CNS-0322615 and IIS-0430224.

References

- [1] G. A. Alvarez, W. A. Burkhard, and F. Christian, "Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering," *Proc. of the 24th Annual Symposium on Computer Architecture* pgs. 62-72, 1997.
- [2] T. E. Anderson, D.E. Culler and D.A. Patterson, "A Case for NOW (Networks of Workstations)," *IEEE Micro*, 15(1), 54-64, 1995.
- [3] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli and R. Wang, "Serverless Network File Systems", *ACM Trans. on Computer Systems*, 41-79, Feb. 1996.
- [4] A. Bhide, E. Elnozahy and S. Morgan, "A Highly Available Network File Server", *Proc. of the Winter 1991 USENIX Technical Conf.*, 199-205, Jan. 1991.
- [5] M. Blaum, J. Brady, J. Bruck and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. on Computers*, 44(2), 192-202, Feb. 1995.
- [6] M. Blaum, J. Brady, J. Bruck, J. Menon, and A. Vardy, "The EVENODD code and its generalization," in *High Performance Mass Storage and Parallel I/O*, pp. 187-208. John Wiley & Sons, INC., 2002.
- [7] M. Blaum, J. Bruck, and A. Vardy, "MDS array codes with independent parity symbols," *IEEE Trans. Information Theory*, vol. 42, no. 2, pp. 529-542, Mar. 1996.
- [8] M. Blaum, R. M. Roth, "New Array Codes for Multiple Phased Burst Correction," *IEEE Trans. on Information Theory*, 39(1), 66-77, Jan. 1993.
- [9] M. Blaum, R. M. Roth, "On Lowest-Density MDS Codes," *IEEE Trans. on Information Theory*, 45(1), 46-59, Jan. 1999.
- [10] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," Technical Report No. TR-95-048, ICSI, Berkeley, California, Aug. 1995.
- [11] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, <http://www.icsi.berkeley.edu/~luby/cauchy.tar:uu>
- [12] V. Bohossian, C. Fan, P. LeMahieu, M. Riedel, L. Xu and J. Bruck, "Computing in the RAIN: A Reliable Array of Independent Node", *IEEE Trans. on Parallel and Distributed Systems*, Special Issue on Dependable Network Computing, 12(2), 99-114, Feb. 2001.
- [13] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance", *Operating Systems Review*, ACM Press, NY, 173-186, 1999.
- [14] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, D. A. Patterson, "Raid - High-Performance, Reliable Secondary Storage," *ACM Computing Surveys*, 26(2), 145-185, 1994.
- [15] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong and S. Sankar, "Row-Diagonal Parity for Double Disk Failure Correction", *Proc. of USENIX FAST 2004*, Mar. 31 to Apr. 2, San Francisco, CA, USA.

- [16] C. Fan and J. Bruck, "The Raincore API for Clusters of Networking Elements", *IEEE Internet Computing*, 5(5), 70-76, Sep./Oct., 2001.
- [17] P. G. Farrell, "A Survey of Array Error Control Codes," *ETT*, 3(5), 441-454, 1992.
- [18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung "The Google File System", Proc. of 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003, pp. 29 - 43
- [19] G. A. Gibson and R. van Meter, "Network Attached Storage Architecture", *Communications of the ACM*, 43(11), 37-45, Nov. 2000.
- [20] G. A. Gibson, D. Stodolsky, F. W. Chang, W. V. Courtright II, C. G. Demetriou, E. Ginting, M. Holland, Q. Ma, L. Neal, R. H. Patterson, J. Su, R. Youssef and J. Zelenka, "The Scotch Parallel Storage Systems," *Proceedings of the IEEE CompCon Conference*, 1995.
- [21] A. V. Goldberg and P. N. Yianilos, "Towards an Archival Intermemory", *Proc. of IEEE Advances in Digital Libraries*, Apr. 1998.
- [22] R. M. Goodman, R. J. McEliece and M. Sayano, "Phased Burst Error Correcting Arrays Codes," *IEEE Trans. on Information Theory*, 39, 684-693, 1993.
- [23] J. H. Hartman and J. K. Ousterhout, "The Zebra Striped Network File System," *ACM Transactions on Computer Systems*, 13(3), 274-310, 1995.
- [24] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, "OceanStore: An Architecture for Global-Scale Persistent Storage", *Proc. of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.
- [25] E. Lee and C. Thekkath, "Petal: Distributed Virtual Disks", *Proc. ACM ASPLOS*, 84-92, Oct. 1996.
- [26] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*, Amsterdam: North-Holland, 1977.
- [27] R. J. McEliece, D. Sarwate, "On sharing secrets and Reed-Solomon codes", *Comm. ACM*, 24(9), 583-584, 1981.
- [28] J. Ousterhout, A. Cherenon, F. Douglass, M. Nelson and B. Welch, "The Sprite Network Operating System", *IEEE Computer*, 21(2): 23-26, Feb. 1988.
- [29] Chong-Won Park and Jin-Won Park, "A multiple disk failure recovery scheme in RAID systems," *Journal of Systems Architecture*, vol. 50, pp. 169-175, 2004.
- [30] J. S. Plank, "A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems," *Software: Practice and Experience*, vol. 27, no. 9, pp. 995-1012, Jan. 1999.
- [31] J. S. Plank and Y. Ding "Note: Correction to the 1997 Tutorial on Reed-Solomon Coding" *Software, Practice & Experience*, vol. 35, no. 2, pp. 189-194, Feb. 2005.
- [32] J. S. Plank, R. L. Collins, A. L. Buchsbaum and M. G. Thomason, "Small Parity-Check Erasure Codes - Exploration and Observations," *International Conference on Dependable Systems and Networks (DSN)*, Yokohama, Japan, Jun. 2005.
- [33] J. S. Plank, M. and T. Moore, "Logistical Networking Research and the Network Storage Stack," *USENIX FAST 2002, Conference on File and Storage Technologies*, work in progress report, January, 2002.
- [34] M. Rabin, "Efficient Dispersal of Information for Security, Load Balancing and Fault Tolerance", *J. ACM*, 32(4), 335-348, Apr. 1989.
- [35] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields", *J. SIAM*, 8(10), 300-304, 1960.
- [36] M. Satyanarayanan, "Scalable, Secure and Highly Available Distributed File Access", *IEEE Computer*, 9-21, May 1990.
- [37] M. Satyanarayanan, J.J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel and D. C. Steere, "CODA - A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, 39(4), 447-459, 1990.
- [38] A. Shamir, "How to Share a Secret", *Comm. ACM*, 612-613, Nov. 1979.
- [39] SUN Microsystems, Inc. *NFS: Network File System version 3 Protocol Specification*, Feb. 1994.
- [40] Chih-Shing Tau and Tzone-I Wang, "Efficient parity placement schemes for tolerating triple disk failures in RAID architectures," in *Proceedings of the 17th International Conference on Advanced Information Networking and Applications (AINA'03)*, Xi'an, China, mar 2003.
- [41] M. Waldman, A. D. Rubin and L. F. Cranor, "Publius: A robust, tamper-evident, censorship-resistant, web publishing system", *Proc. 9th USENIX Security Symposium*, 59-72, Aug. 2000. Online at: <http://www.cs.nyu.edu/~waldman/publius/publius.pdf>
- [42] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliccote and P. K. Khosla, "Survivable Information Storage Systems", *IEEE Computer*, 33(8), 61-68, Aug. 2000.
- [43] L. Xu and J. Bruck, "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Trans. on Information Theory*, 45(1), 272-276, Jan., 1999.
- [44] L. Xu, V. Bohossian, J. Bruck and D. Wagner, "Low Density MDS Codes and Factors of Complete Graphs," *IEEE Trans. on Information Theory*, 45(1), 1817-1826, Nov. 1999.

WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems

James Lee Hafner
IBM Almaden Research Center
hafner@almaden.ibm.com

Abstract

We present the WEAVER codes, new families of simple highly fault tolerant XOR-based erasure codes for storage systems (with fault tolerance up to 12). The design features of WEAVER codes are (a) placement of data and parity blocks on the same strip, (b) constrained parity in-degree and (c) balance and symmetry. These codes are in general not maximum distance separable (MDS) but have optimal storage efficiency among all codes with constrained parity in-degree. Though applicable to RAID controller systems, the WEAVER codes are probably best suited in dRAID systems (distributed Redundant Arrangement of Independent Devices). We discuss the advantages these codes have over many other erasure codes for storage systems.

1 Introduction

It has become increasingly clear in the storage industry that RAID5 does not provide sufficient reliability against loss of data either because of multiple concurrent disk losses or disk losses together with sector losses (e.g., due to medium errors from the disks). The reasons are primarily due to the dramatic increase in single disk capacity together with a fairly constant per-bit error rate. Additional factors, as mentioned in [5], include increasing number of disks per system, and use of less reliable disks such as ATA (vs. SCSI). The cited paper makes the case for double fault tolerance; by extrapolation, there is (or will be) a need for higher fault tolerant codes if these trends continue. Furthermore, as the industry moves into very long-term archival storage or dRAID (distributed Redundant Arrangement of Independent Devices) node-based systems, the need for higher fault-tolerance erasure codes will likely become more acute.

N -way mirroring can clearly be used to provide additional redundancy in any system, but the storage efficiency (ratio of user data to the total of user data plus redundancy data) of mirroring is very low. (We prefer the term “efficiency” in storage contexts instead of the equivalent term “rate” which is more suitable for communication channels.) On the other hand, codes like Reed-Solomon (RS) [14] provide optimal storage efficiency (that is, are maximum distance separable, or MDS) and arbitrarily high fault tolerance, but require

special purpose hardware to enable efficient computation of the finite field arithmetic on which the codes are based (or, if formulated as a binary XOR code, generally have higher computational costs and complexities).

Other erasure codes have been proposed for better fault tolerance than RAID5, but none has emerged as a clear winner even in the RAID controller market – the industry has not even settled on a *de facto* standard for 2 fault tolerance after 40+ years (since RS codes were first proposed). We believe no such “perfect” code can exist; every code requires some trade-offs in efficiency, performance or fault tolerance.

In this paper, we present the WEAVER codes, so called because the parity/redundancy values are computed by XOR formulas defined by patterns that weave through the data. There are three design principles that characterize WEAVER codes: (a) every strip (stripe unit) contains both data and parity from the same stripe (we call these vertical codes because data and parity are arranged together vertically on each strip), (b) the number of data values that contribute to each parity value (parity in-degree) is fixed and, most importantly, is independent of the stripe size (number of strips) and (c) balance and symmetry. The second property enables flexibility in choices of stripe sizes without altering computational costs (in both XOR and IO). In addition, it bounds the computational costs of many operations (e.g., short writes, rebuild). More details on these points are given in Section 3 and elsewhere.

The WEAVER codes are designed with balance and symmetry in three aspects. First, every parity is constructed from some fixed number of data values (we call this number the “parity in-degree” – as noted, it is independent of the stripe size). Second, each data value contributes to a fixed number of parity values (we call this the “data out-degree”; it is also independent of the stripe size). The data out-degree for WEAVER codes is set to the fault tolerance, the theoretical minimum number for the given fault tolerance. For additional symmetry, some of the codes have the parity in-degree equal to the data out-degree – this provides a certain duality between data and parity. Third, all the code constructions are specified by a weave-pattern which is repeated by simple rotation of a base configuration. That is, they are rotationally symmetric.

We bound the parity in-degree by the fault tolerance to control the complexity of parity computations and improve other properties such as localization (see Section 3.1.2). There are undoubtedly other whole families of codes yet to be discovered that relax this requirement, while keeping the parity in-degree fixed (some Wiencko codes [15, 6] may have this property as well).

There are two general families and one ad-hoc family of WEAVER codes, which we describe in detail later. We briefly mention here that there are constructions of WEAVER codes that tolerate up to 12 device failures (and perhaps beyond). A key feature of all WEAVER codes is the “localization” property that for large stripe sizes limits the scope of most operations (including, for example, rebuild) to small subsets of the stripe. This is discussed in more detail in Section 3.1.2.

The WEAVER codes are in general *not* MDS codes (though some special cases are). Consequently, the main disadvantage of these codes is their storage efficiency. However, these codes are optimally efficient for the given fault tolerance and parity in-degree constraint (see Section 3.2). Of the three families of WEAVER codes, one family has efficiency 50% for all levels of fault tolerance (up to 10 in our constructions); the other families have lower efficiency which decreases with increasing fault tolerance. In all cases, these codes have significantly higher efficiency than N -way mirroring. The WEAVER codes, by their symmetry, have a certain simplicity of implementation (though not as simple as N -way mirroring). As such these codes provide a way for a system designer to select highly fault tolerant codes that interpolate between N -way mirroring with its performance advantages, exceptional simplicity but minimal efficiency and MDS codes with somewhat lower performance and somewhat greater complexity but optimal efficiency.

Unfortunately, we do not have many theoretical results concerning specific constructions. Generally, for small fault tolerance, these codes can be tested by hand (in some cases, we give the proof). For other cases, our constructions were tested by computer program using the generator matrix (see Section 4 and [10] for related methodology).

The paper is organized as follows. We close this introduction with some definitions and notation. In Section 2 we describe the various constructions for each family of WEAVER codes. Section 3 lists the key advantages and disadvantages of these codes. Related work and comparisons with other published codes are discussed in detail in Section 5. Section 4 outlines our testing methodology. We conclude with a short summary.

1.1 Vocabulary and Notations

The literature contains some inconsistency concerning the use of common storage and erasure code terms, so we state our definitions here to avoid confusion. We use the term “system” to refer to either a dRAID storage system of node-type devices or to a controller array of disks (RAID). The term “device” will refer to the “independent” storage device in the system (a node in dRAID or a disk in RAID).

element: a fundamental unit of data or parity; this is the building block of the erasure code. In coding theory, this is the data that is assigned to a bit within the symbol. For XOR-based codes, this is typically one or more sequential sectors on a disk (or logical sectors on a storage node).

stripe: a complete (connected) set of data and parity elements that are dependently related by parity computation relations. In coding theory, this is a code word; we use “code instance” synonymously.

strip: a unit of storage consisting of all contiguous elements (data and/or parity) from the same device and stripe (also called a stripe unit). In coding theory, this is a code symbol. The set of strips in a code instance form a stripe. Typically, the strips are all of the same size (contain the same number of elements).

vertical code: an erasure code in which a (typical) strip contains both data elements and parity elements (e.g., X-code [17] or these WEAVER codes). Contrast this notion with a “horizontal code” in which each strip contains either data elements or parity elements, never both (e.g., EVENODD [2]).

We use the symbol t exclusively to represent the fault tolerance of a code, the symbol n for the size of the stripe (the number of strips, or equivalently, the number of devices in a code instance), and k for the maximum parity in-degree. For WEAVER codes, all parity have in-degree exactly k and $k \leq t$. In addition, all data have out-degree equal to t . We therefore parameterize our codes as $\text{WEAVER}(n, k, t)$, and we provide constructions for different values of these parameters. We let r denote the number of data elements and q the number of parity elements per strip. (We see how r and q may be determined from k and t in Section 2.)

We define a “short write” as a host write to any sequential subset of an element (e.g., a single sector); a “multiple strip write” as a host write to a sequential subset of the strips in a stripe (that is, the user data portion of the strips). The “write lock zone” is the set of elements that should be locked during a short write so as to provide data/parity consistency in case of failures encountered during a write operation. The “rebuild zone” is the subset of strips within the stripe which are needed during

a rebuild of one or more lost strips. We see why these notions are relevant to WEAVER codes in Section 3.

2 WEAVER code definitions

In this section we describe our WEAVER constructions. We use a graph representation to visualize the constrained parity-in degree and fixed data-out degree. A table format is used to visualize the data and parity layout on strips (and so devices). Formulas for “parity defining sets” (see below) are used to precisely define each construction. Each of these presentations exhibit some of the balance and symmetry of the WEAVER codes.

Figure 1 shows a directed, bipartite graphical representation of a general WEAVER code; the nodes on top represent the data elements in the stripe and the nodes on the bottom represent the parity elements. An edge connects a data element to a parity element if that data element contributes to the parity value computation (we say that the data element “touches” this parity element). For $\text{WEAVER}(n,k,t)$ codes, each data element has data out-degree equal to t . In addition, each parity element has parity in-degree exactly k , where $k \leq t$. As mentioned, this constraint is the key to many of the good properties of these codes.

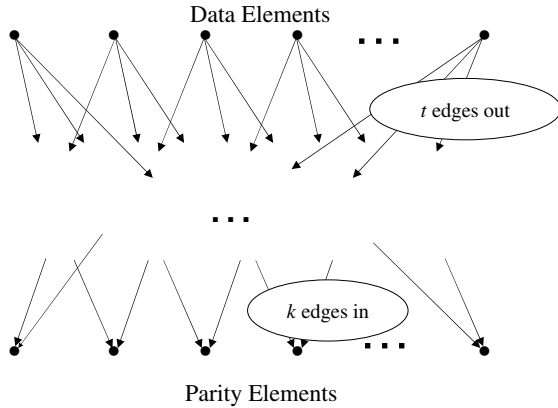


Figure 1: Graphical representation of a general $\text{WEAVER}(n,k,t)$ code. Each parity element has in-degree equal to k ; each data element has out-degree equal to t .

Generally, a graphical representation like that of Figure 1 can be used for any XOR-based erasure code (for example, see the Tornado codes [11]). In addition, Tanner graphs may also be used (see the description of the Low-Density Parity-Check codes in [13, 12]). In Tanner graphs the nodes on one side represent data and parity and the opposite nodes represent parity checks. The systematic and regular nature of our WEAVER codes makes the Tanner representation less useful for visualization. We use our graphs to show the encoding of data/parity relations, and not for decoding as in [11]. We also draw

our graph with nodes on top and bottom (not left/right) to suggest a relationship to the data/parity layout onto strips as described next.

Figure 1 only provides a partial description of the code in the context of storage systems. The physical layout of data and parity on the strips within the stripe must also be specified. This is given in Figure 2 where we see the vertical nature of the WEAVER codes. As noted, other codes share this vertical layout (see, for example, the X-code [17] and the BCP code [1]). We view the logical addressing of the data elements from the host’s viewpoint as first within a strip and then strip to strip.

S_0	S_1	\dots	S_j	\dots	S_{n-1}
$d_{0,0}$	$d_{0,1}$	\dots	$d_{0,j}$	\dots	$d_{0,n-1}$
$d_{1,0}$	$d_{1,1}$	\dots	$d_{1,j}$	\dots	$d_{1,n-1}$
\vdots	\vdots	\dots	\vdots	\vdots	\vdots
$d_{r-1,0}$	$d_{r-1,1}$	\dots	$d_{r-1,j}$	\dots	$d_{r-1,n-1}$
$p_{0,0}$	$p_{0,0}$	\dots	$p_{0,j}$	\dots	$p_{0,n-1}$
$p_{1,0}$	$p_{1,0}$	\dots	$p_{1,j}$	\dots	$p_{1,n-1}$
\vdots	\vdots	\dots	\vdots	\vdots	\vdots
$p_{q-1,0}$	$p_{q-1,0}$	\dots	$p_{q-1,j}$	\dots	$p_{q-1,n-1}$

Figure 2: Stripe/strip layout of general $\text{WEAVER}(n,k,t)$ code. Each strip contains r data elements and q parity elements. S_j denotes a strip label; $d_{i,j}$ is a labeled data element; $p_{i,j}$ is a labeled parity element. Each strip is stored on a different device in the system.

Of course, there is no requirement that the parity elements be placed below the data elements; they must be placed however on the same device. WEAVER codes always have both data and parity elements on each strip. Essential for performance is that the data elements be logically contiguous, as must the parity elements.

We can also represent our codes by sets of indices. A parity element $p_{i,j}$ can be relabeled $p_{\kappa(i,j)}$ where $\kappa(i,j)$ is the set of (ordered pair) indices of the data elements that touch this parity element. That is,

$$p_{i,j} \rightarrow p_{\kappa(i,j)} = \bigoplus_{(u,v) \in \kappa(i,j)} d_{u,v}. \quad (1)$$

Conversely, we can relabel the data elements as $d_{\tau(i,j)}$ where $\tau(i,j)$ is the set of (ordered pair) indices of the parity elements that are touched by $d_{i,j}$. That is,

$$\tau(i,j) = \{(u,v) : (i,j) \in \kappa(u,v)\}. \quad (2)$$

In the graph of Figure 1, $\kappa(i,j)$ can label the set of edges into parity node $p_{i,j}$, and, similarly, $\tau(i,j)$ can label the set of edges out of data node $d_{i,j}$. This notation is used to provide formulas to define specific constructions.

For notation, define for any set of ordered pairs of indices κ and any offset s , a new set of indices referred to as $\kappa + s$ using the operation

$$\kappa + s \stackrel{\text{def}}{=} \{(u, v + s \bmod n) : (u, v) \in \kappa\},$$

so each column index in $\kappa + s$ is offset by s modulo n .

Using this notation, we provide rotational symmetry (a key design feature of WEAVER codes). For $0 \leq j \leq n-1$ and $0 \leq i \leq q-1$, set

$$\kappa(i, j) = \kappa(i, 0) + j. \quad (3)$$

In other words, a specification for the parities $p_{\kappa(i,0)}$ for $0 \leq i \leq q-1$ (on the first strip), together with rotation to the right (with wrap-around) provides a complete specification of WEAVER(n, k, t) erasure codes. We call the sets $\kappa(i, j)$ “parity defining sets”. A similar rotational formula can be derived for the sets $\tau(i, j)$.

By counting edges in two ways, it is easy to see from Figures 1 and 2 that $rt = qk$. Generally, a WEAVER(n, k, t) code will have r and q minimal (to minimize overall complexity): so $r = k/m$ and $q = t/m$ where $m = \gcd(k, t)$. This is assumed throughout unless otherwise noted.

Given these parameters, the storage efficiency for these codes is given by

$$\text{Eff} = \frac{nr}{nr + nq} = \frac{r}{r + q} = \frac{k}{k + t}. \quad (4)$$

The first two are obvious from Figure 2, the latter comes from the relation $r = qk/t$. Since we assume $k \leq t$, the maximum efficiency for any WEAVER code is 50%.

In the next few subsections, we describe specific constructions of parity defining sets that provide for prescribed fault tolerance.

2.1 WEAVER codes of efficiency 50%

For our first family of WEAVER codes we set $k = t$ so that efficiency is 50%. We have $\gcd(k, t) = k = t$ so that $r = q = 1$ and the layout of Figure 2 has only one row of data and one row of parity (see the example below). For this family of codes, we suppress the first component of each index pair and refer to our parity defining sets simply as $\kappa(j) = \kappa(0, j)$ for $0 \leq j \leq n-1$. We use the following additional notation. Let $\kappa_1(i)$ be an increasing sequence of k integers with initial value 1 and let s be an “offset”. We can specify a parity defining set $\kappa(j)$ from $\kappa_1(j)$ and s by the relation

$$\kappa(j) \stackrel{\text{def}}{=} \kappa_1(j) + s = \{i + s \bmod n : i \in \kappa_1(j)\}.$$

As we will see, simple $\kappa_1(j)$ sets, together with different offsets s provide a convenient way to specify good parity defining sets. As before, if we impose rotational

symmetry (see equation (3)), then we need only specify $\kappa_1(0)$ and s to completely determine the code. We overload the term “parity defining set” to include a set $\kappa_1(0)$ and an offset s .

For example, with $\kappa_1(0) = \{1, 2, 4\}$ and $s = 2$ (see Table 1, $t = 3$, second entry), the following diagram provides a valid WEAVER code provided $n \geq 7$.

S_0			S_j		S_{n-1}
d_0	d_1	\cdots	d_j	\cdots	d_{n-1}
$p_{\{3,4,6\}}$	$p_{\{4,5,7\}}$	\cdots	$p_{\{j+3,j+4,j+6\}}$	\cdots	$p_{\{2,3,5\}}$

Table 1 gives a partial listing of parity defining sets ($\kappa_1(0)$ and offset s) and valid stripe sizes n for fault tolerance $1 \leq t \leq 10$. (We say a stripe size is “valid” for a given parity defining set if the code on that stripe size has the required fault tolerance.) The entries tagged with an asterisk are discussed in the remarks below.

We make the following remarks concerning Table 1.

- The first entry in the table is a simple RAID1 mirror code, but with a non-standard data layout. This code easily and uniformly provides simple mirroring on any number of devices (at least 2), including an odd number of devices. In addition it provides load-balancing; every device is equally burdened by data and a parity mirror. See Section 5 for further comments on the $t = 2$ entry of the table.

- The first three rows in the table show valid codes for $t \leq 3$ with $n \geq 2t$. When $n = 2t$, these codes are in fact MDS. For larger t we could not find constructions that maintained this property. One can measure the “space efficiency penalty” as the difference between the optimal efficiency of an MDS codes on n strips and the actual efficiency; for these WEAVER codes, this is:

$$\frac{n-t}{n} - \frac{1}{2} = \frac{1}{2} - \frac{t}{n}.$$

As can easily be calculated, this ranges from 0.0 to 0.23 for the values in the table, using the smallest valid n for each t . It also increases as n increases for a fixed fault tolerance t (but larger n improves the localization effects as in Section 3.1.2).

- The table provides only a small subset of all the constructions we discovered. For this work, we tested validity for all cases of stripe sizes n , offsets s and parity defining sets $\kappa_1(0) \subseteq [1, w]$ of various ranges. See Section 4 for the methodology we used to test configurations. For $t \leq 7$, we covered the ranges $n \leq t^2 + 2t$, $0 \leq s \leq 8$ and $w = 3t$. For $t = 8$, we ran a preliminary filter to find good candidate parity defining sets, then processed the most promising ones in the range $n \leq t^2 = 64$, $0 \leq s \leq 8$ and $w = 2t = 16$. For $t = 9, 10$, we did a preliminary search with $0 \leq s \leq 8$ and $w = 2t$, with $n \leq 4t + 4$ (see the next remark) and

t	$\kappa_1(0)$	offset s	Stripe Size n
1	$\{1\}^*$	0	2+
2	$\{1, 2\}^*$	0	4+
3	$\{1, 2, 3\}^*$	1	6,8+
	$\{1, 2, 4\}$	2	7+
4	$\{1, 3, 5, 6\}$	1	10+
	$\{1, 2, 3, 6\}^*$	0,2,3	11+
5	$\{1, 3, 4, 5, 7\}$	2	12,15+
	$\{1, 5, 6, 8, 9\}$	3	13+
	$\{1, 2, 3, 4, 7\}$	1	14+
	$\{1, 2, 3, 6, 9\}^*$	2	15+
6	$\{1, 5, 8, 9, 10, 12\}$	2	17,19,21+
	$\{1, 6, 8, 9, 11, 12\}$	7	17,20+
	$\{1, 2, 3, 6, 9, 10\}$	0	18+
	$\{1, 2, 3, 4, 6, 9\}^*$	5	19+
7	$\{1, 4, 5, 6, 7, 8, 11\}$	4	20,23-24,26,28+
	$\{1, 2, 4, 7, 10, 12, 13\}$	3	20,24+
	$\{1, 3, 5, 6, 7, 11, 12\}$	1	22+
	$\{1, 2, 3, 4, 6, 9, 14\}^*$	6	23+
8	$\{1, 2, 4, 8, 10, 11, 12, 13\}$	0	26,28+
	$\{1, 2, 6, 7, 8, 9, 12, 14\}$	0	27+
	$\{1, 2, 3, 4, 6, 7, 9, 14\}^*$	0	28+
9	$\{1, 4, 5, 6, 7, 12, 13, 15, 18\}$	2	30,32,34+
	$\{1, 4, 5, 8, 11, 12, 13, 14, 15\}$	6	31+
	$\{1, 2, 3, 4, 6, 7, 9, 14, 15\}^*$	5	32+
10	$\{1, 2, 5, 6, 7, 10, 13, 15, 19, 20\}$	3	35,40+
	$\{1, 2, 4, 7, 8, 9, 13, 14, 17, 18\}$	0	37+
	$\{1, 2, 3, 4, 6, 7, 9, 14, 15, 19\}^*$	3	40+

Table 1: Partial listing of parity defining sets for WEAVER(n, t, t) codes. See the remarks for a description of the entries tagged with an asterisk. A stripe size n_0+ means $n \geq n_0$.

then verified the table entries up to $n \leq 64$ for $t = 9$ and $n \leq 56$ for $t = 10$. For $t \leq 3$, it is fairly easy to prove that the constructions work for all n in the described range. For $t \geq 4$, the implication that the codes work for n outside the tested range is not theoretically established; see Section 6 and Theorem 1 in particular. Note that, theoretically, we could have extended the offset range up to $n - w - t$, but that would have made our search spaces even larger, particularly for large n (we did limit it for small n when appropriate).

- The search space for these experiments is actually quite large. For a given t and w , there are $\binom{w-1}{t-1}$ sets $\kappa_1(0)$ and 9 offsets s each to examine. For each such parity defining set and each n , there are up to $\binom{n-1}{t-1}$ individual matrix rank tests to perform (see Section 4). These numbers grow rapidly with higher fault tolerance. For example, the $t = 6$ search completed at least 1.6 trillion matrix rank tests. For $t \geq 6$ it was prohibitive to do this on a standard workstation. Instead, we implemented our search to run on a portion (only 1024 processors) of an IBM Blue Gene/L system. Each processor was given a subcollection of the search space of sets $\kappa_1(0)$ and, for each $\kappa_1(0)$ in its subcollection, ran the tests for every

offset s and n in the ranges mentioned above. The $t = 6$ case mentioned above took approximately 12 hours on 1024 processors. The $t = 10$ preliminary search (for $n \leq 44$) took approximately 37.6 days and completed more than 64 trillion matrix rank tests!.

- For fault tolerance $t \geq 4$, there are gaps in the sequence defining the set $\kappa_1(0)$. This is a requirement as the following argument illustrates. Suppose $k = t = 4$ and there are four consecutive integers in $\kappa(0)$, say, $i, i+1, i+2, i+3$. Consider the data element labels $i+1$ and $i+2$. Both appear together in $\kappa(n-1) = \kappa(-1)$, $\kappa(0)$ and $\kappa(1)$. But $i+1$ appears by itself in $\kappa(2)$ and $i+2$ appears by itself in $\kappa(-2)$. If we lose strips $i+1$ and $i+2$ (so data elements d_{i+1} and d_{i+2}), and strips 2 and $(n-2)$ (with $p_{\kappa(2)}$ and $p_{\kappa(-2)}$), then every surviving parity contains either both of the data elements $i+1$ and $i+2$ or neither. Consequently, there is no way to distinguish between the two values and this 4 strip failure case cannot be tolerated. It is not clear what additional heuristics (or theorems) define “good” or “bad” sets $\kappa(0)$ (see Section 6).

- We listed only entries that have valid stripe sizes for all $n \geq n_0$ (identified in the table as n_0+), with perhaps

a few isolated valid stripe sizes below n_0 . For example, for $t = 5$, the entry $\kappa_1(0) = \{1, 3, 4, 5, 7\}$ with offset $s = 2$ has valid stripe sizes $n = 12$ and $n \geq 15$ but not $n = 13, 14$. We typically observed similar behavior for almost all sets we tested though there were anomalies. The set $\kappa_1(0) = \{1, 3, 6, 10, 15, 21\}$ had invalid stripe sizes for every n divisible by 9 regardless of offset. We do not have a proof that this persists, but we believe that a proof of such negative results would not be difficult.

- The entries marked with * form a single chain of supersets for $\kappa_1(0)$ as t increases. The usefulness of this chain is described in more detail in Section 3.1.4, but briefly it enables changing of fault tolerance on-the-fly with minimal parity recomputation.

2.1.1 The special case of 2 fault tolerance

Consider the $t = 2$ element from Table 1 where $\kappa(0) = \kappa_1(0) = \{1, 2\}$ (and $s = 0$). We will describe this code in somewhat different terms and prove the claimed fault tolerance. With $k = t = 2$, each parity value is the XOR sum of a pair of data values, one from each of the two strips immediately to the right of the parity element (with wrap-around, of course). Alternatively, each data element touches two parity elements and so is paired with two other data elements (the other data element in each of its two parity elements). From this viewpoint, a given data element D is paired with its west W and east E neighbor elements (graphically):

$$W \leftarrow D \rightarrow E$$

The parity value computed from $W \oplus D$ is stored in the strip to the left of W ; the parity value computed from $D \oplus E$ is stored on the strip to the left of D , namely, the same strip as W .

We now give a proof that this code has the required 2 fault tolerance, provided $n \geq 4$. It is clear that this is a necessary condition (if $n \leq 3$ and two strips are lost, there is at most only one strip remaining and that is clearly insufficient). It is also clear that we only need to recover lost data values, as parity values can be recomputed from all the data.

There are three cases to consider.

Case 1: Suppose one strip is lost (say, strip S_j). We observe that $\kappa(j-1) = \{j, j+1\}$ so that d_j can be recovered by reading d_{j+1} (from the strip S_{j+1} to the right of strip S_j), reading $p_{\kappa(j-1)}$ (from the strip S_{j-1} to the left of S_j), and using the formula:

$$d_j = d_{j+1} \oplus p_{\kappa(j-1)}. \quad (5)$$

Alternatively, we can recover d_j from d_{j-1} and $p_{\kappa(j-2)}$ since $\kappa(j-2) = \{j-1, j\}$.

Case 2: If two adjacent strips are lost (say, S_j and S_{j+1}), we read d_{j-1} and $p_{\kappa(j-1)}$ (in one operation from

strip S_{j-1}) and $p_{\kappa(j-2)}$ from S_{j-2} . Then, recursively,

$$\begin{aligned} d_j &= p_{\kappa(j-2)} \oplus d_{j-1} \\ d_{j+1} &= p_{\kappa(j-1)} \oplus d_j. \end{aligned}$$

Note that even though we needed to read three elements (two parity and one data), we only needed to access two devices (for disks, this is a single IO seek per device) because of the vertical arrangement of data and parity.

Case 3: If two non-adjacent strips are lost, then we reconstruct as two independent cases of a single strip loss (using (5)), because the data can always be reconstructed from its left neighboring parity and right neighboring data (neither of which is on a lost strip). For $n \geq 4$, most of the dual failure cases are of this type.

In this proof, we saw two examples of the “localization” property of WEAVER codes (see Section 3.1.2). For all cases, only a few devices in the stripe need to be accessed for reconstruction; this number is independent of the stripe size n . In addition, in Case 3 the reconstruction problem split into two smaller, easier and independent reconstruction problems.

We also saw how the vertical layout reduces device access costs, by combining some multi-element reads into a single device access. For disk arrays, this implies fewer disk seeks and better performance.

2.2 Other constructions with one data row

The constructions of the previous section are “ad hoc”; that is, (with the exception of the case $t = 2, 3$) they were found by computer search and not by a parametric formulation of the parity defining sets $\kappa(0)$. In this section, we give a different and formulaic construction.

We start by making the assumption that k divides t . Then $\gcd(k, t) = k$ so we can take $r = 1$ (one data row) and $q = t/k$ (q parity rows). We use the term consecutive- i to mean consecutive numbers with difference i . (For example, the consecutive-2 numbers starting at 1 are 1, 3, 5, ...) We say a set of parity elements are consecutive- i if they are in the same row and their corresponding strip numbers are consecutive- i , modulo n .

The constructions are described as follows. In parity row 0, data element d_j touches k consecutive-1 parity elements to the left, ending at some left-offset s from d_j (with wrap-around). In each parity row i , $1 \leq i \leq q-1$, data element d_j touches the set of k consecutive- $(i+1)$ parity elements ending one strip to the left of the first parity element touched in the previous row (again, with wrap-around). A data element touches exactly k parity elements in each row so that each parity is composed of k data elements (that is, parity in-degree is k).

Figure 3 shows a graph representation for the special case when $k = 3$, $t = 9$ and so $r = 1$ and $q = 3$. This graph is that subgraph of the general graph in Figure 1

corresponding to the out-edges for the one data element d_j (and by rotational symmetry, implies the subgraph for all other data elements).

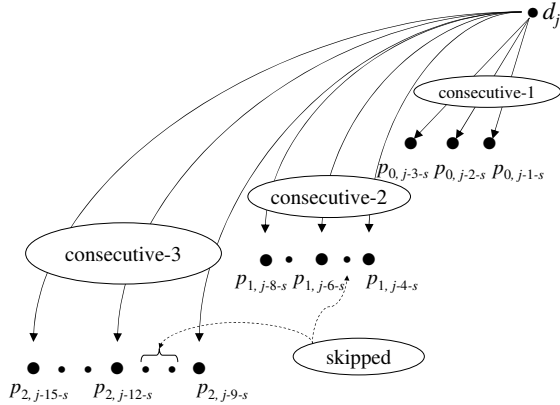


Figure 3: The subgraph of the data/parity relations graph for $k = 3$, $t = 9$ and offset s . The relative position suggests approximate placement in the rows of the data/parity layout of Figure 2. Small dots represent parity elements in the gaps that not touched by d_j .

Put in mathematical terms, the set $\tau(j) = \tau(0, j)$ of (ordered pairs of) indices of parity elements touched by $d_j = d_{0,j}$ is given by the formula

$$\tau(j) = \bigcup_{i=0}^{q-1} \{(i, \langle j - \sigma(i, k, u) - s \rangle_n) : 1 \leq u \leq k\}. \quad (6)$$

where, as shorthand, $\langle x \rangle_n = x \bmod n$ and

$$\sigma(i, k, u) = (k-1)i(i+1)/2 + u(i+1).$$

From the parity viewpoint, the equivalent formulation is

$$\kappa(i, j) = \{\langle j + \sigma(i, k, u) + s \rangle_n : 1 \leq u \leq k\}. \quad (7)$$

In these expressions, the term $u(i+1)$ for $1 \leq u \leq k$ provides the k consecutive- $(i+1)$ parity elements. The term s provides the initial offset. The term $(k-1)i(i+1)/2 + s$ provides for the starting point relative to d_j .

Table 2 provides a list of some examples found by testing all $s \leq 8$ and all $n \leq 64$ for $t \leq 10$ and $n \leq 48$ for $t = 12$. We also give the efficiency, $\text{Eff} = 1/(q+1)$ by (4); in the previous section, all codes had efficiency 50%. Notice the examples with fault tolerance as high as $t = 12$. We conjecture that this construction (with a suitable offset s and sufficiently large n) should work for arbitrarily large t , though not necessarily all k .

The examples with $t = 9, k = 3$ and $t = 12, k = 4$ are interesting because the fault tolerances are so high, but the efficiency is 250% higher and 325% higher than

t	k	q	s	Stripe Size n	Efficiency
2	2	1	0,1	4+	50%
4	2	2	0	6+	33%
6	2	3	0,1,2	9+, excl. 9+s	25%
8	2	4	0,1,2,3	15+, excl. 14+s	20%
10	2	5	0,3,4	16,20+, excl. 20+s	17%
12	2	6	0,1	23,27+, excl. 27+s	14%
3	3	1	1	6,8+	50%
6	3	2	2,4 3	11,13,15+ 12,14+	33%
9	3	3	1 3	15,17+ 16-17,19+	25%
12	3	4	1,3	24+, excl. 24+s	20%
12	4	3	2	21,25+	25%

Table 2: Partial listing of $\text{WEAVER}(n, k, t)$ codes where k divides t and parity defining sets given by (6) or (7).

corresponding 10-way or 13-way mirrors. The two examples in Table 2 with $t = k = 2$ and $t = k = 3$ are identical to the $\text{WEAVER}(n, t, t)$ codes given in Table 1 with $\kappa_1(0) = \{1, 2\}$ and $\kappa_1(0) = \{1, 2, 3\}$, respectively.

There are two codes in Table 2 with the same fault tolerance $t = 6$; they have different parity in-degree k and so different efficiency. The code with $k = 2$ has very simple parity computations but efficiency only 25%. The code with $k = 3$ has somewhat more complex parity computations but better efficiency 33%. This exemplifies one of the trade-offs of erasure codes (performance vs. efficiency) and the fact that the rich set of WEAVER codes provide a means to balance these trade-offs for a given system's constraints. (Similar remarks apply to the three codes with $t = 12$.)

The remark made in the comments of Section 2.1 about consecutive parity elements when $k = t$ precludes this construction from working for $k = t \geq 4$. For $k = 4$ and $t > 4$ the situation is undecided. Preliminary experiments suggest that $t = 8$ suffers from a similar obstacle; surprisingly $t = 12$ does have some valid configurations. Clearly, alternatives to consecutive-1 may be required if $k \geq 4$ (though we have not tested any of these cases).

There is considerably more structure to the patterns of valid offsets and stripe sizes for this construction compared to those of the previous section. It is likely that this construction can be analyzed in most cases theoretically. We conjecture that other parity defining sets will also provide valid WEAVER codes of similar efficiency and design, perhaps on even smaller stripe sizes or will fill in the gaps in the stripe sizes in Table 2. We leave these two issues to future work.

2.3 Parity in-degree 2

The constructions we have presented so far have one data row ($r = 1$, because k divides t in all cases). In the next

two sections, we give two ad-hoc constructions (with 3 and 4 fault tolerance, respectively) that have $r = 2$. The key feature of these codes is that we restrict $k = 2$, so that the parity computations are quite simple.

These codes generalize the 2 fault tolerant code discussed in Section 2.1.1. They each contain two copies of this 2 fault tolerant code, plus some additional cross relations between them (hence, again the WEAVING name). All three codes use “data neighbor” relations to determine the parity defining sets. Adding a second row allows for relations in different directions than just east/west; this in turn enables more parity defining sets containing a given data element; and this increases the (potential) fault tolerance.

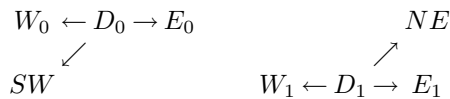
2.3.1 Three fault tolerance

The WEAVING($n,2,3$) code presented in this section has a data/parity layout given by

S_0		S_j		S_{n-1}
$d_{0,0}$...	$d_{0,j}$
$d_{1,0}$...	$d_{1,j}$
$P_{\{(0,1),(0,2)\}}$...	$P_{\{(0,j+1),(0,j+2)\}}$
$P_{\{(1,1),(1,2)\}}$...	$P_{\{(1,j+1),(1,j+2)\}}$
$P_{\{(1,n-2),(0,n-1)\}}$...	$P_{\{(1,j-2),(0,j-1)\}}$

Each parity element is labeled by its parity defining set. The first parity row is the WEAVING($n,2,2$) code built from the first data row. The second parity row is again the WEAVING($n,2,2$) code built from the second data row. The third row weaves between these two along nearest neighbor up-diagonals, placing the parity to the right of the up-neighbor (other placements are possible).

Visually, data elements D_0 from the first row and D_1 from the second row are paired with neighbors as in the following diagram (each pair computes a different parity value):



The $D_1 \rightarrow NE$ relation is just a reverse perspective on the $D_0 \rightarrow SW$ relation above ($SW = D_1$ and $NE = D_0$). Each data element touches three different parity elements on three different strips, twice in one parity row, and once in the last row parity row. This provides the necessary condition for 3 fault tolerance.

It can be proven that this construction is also sufficient for 3 fault tolerance provided $n \geq 6$ but we leave out the proof as it is similar to the one we gave for WEAVING($n,2,2$).

This construction provides another example of a 3 fault tolerant code on as few as 6 devices. Compare this with the first $t = 3$ entry in Table 1 where $k = 3$;

the difference is again a performance/efficiency trade-off. See Section 5 for more comments.

2.3.2 Four fault tolerance

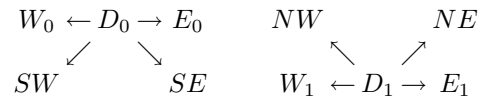
In this section we provide another WEAVING($n,2,4$) code with two data rows. Contrast this construction with the code of Section 2.2, which also has $k = 2$ and $t = 4$ but has only one data row (see Table 2). This is the only code we present that drops the “minimalist” condition $r = k / \gcd(k, t)$.

The layout and parity defining sets can be seen in the following chart:

S_0		S_j		S_{n-1}
$d_{0,0}$...	$d_{0,j}$
$d_{1,0}$...	$d_{1,j}$
$P_{\{(0,1),(0,2)\}}$...	$P_{\{(0,j+1),(0,j+2)\}}$
$P_{\{(1,1),(1,2)\}}$...	$P_{\{(1,j+1),(1,j+2)\}}$
$P_{\{(1,n-3),(0,n-2)\}}$...	$P_{\{(1,j-3),(0,j-2)\}}$
$P_{\{(0,n-3),(1,n-2)\}}$...	$P_{\{(0,j-3),(1,j-2)\}}$

The first five rows are essentially identical to the WEAVING($n,2,3$) code of the previous section, with the exception that the fifth row is cyclically shifted right by one. The last row of parity elements is computed by weaving the down-diagonals and placing the parity value in the parity element two strips to the right.

This time, each data element is paired with four other data elements: west, east, and two diagonal neighbors (southwest, southeast for an element D_0 in row 0 and northeast, northwest for an element D_1 in row 1). Graphically, this looks like:



Each data element is now stored in four parity elements: twice in one of the first two parity rows (and not in the other), and once in each of the last two parity rows.

Again, we leave out the proof that this is 4 fault tolerant provided $n \geq 8$. This code has the same fault tolerance, efficiency and parity in-degree as the WEAVING($n,2,4$) code in Table 2, but requires more strips for the minimal configuration (and has six rows in total versus three for the previous code). However, it has better localization properties. For example, for this construction the write lock zone comprises a total of eight data and parity elements on six neighboring strips (centered on the target strip) versus the same number of data and parity elements on seven strips (five neighboring strips similarly centered plus two additional strips) for the $k = 2, t = 4$ code of Section 2.2.

Note that by dropping the first two rows of parity, we get yet another WEAVING($n,2,2$) code with two data

rows, two parity rows and efficiency 50%. It can be made equivalent to the code described in Section 2.1.1 and so is not very interesting by itself.

3 Features

Now that we have defined the WEAVER codes and given many examples, we next discuss the key features of these codes, the advantages in the next subsection and a brief discussion of the primary disadvantage following.

3.1 Advantages

3.1.1 High Fault Tolerance

The WEAVER codes have instances of exceptionally high fault tolerance (we gave constructions with fault tolerance 12 and conjecture that other constructions should be possible). There are few codes in the storage system literature that meet these fault tolerance levels. The only viable options for very high fault tolerance to date seem to be Reed-Solomon codes, with their high computational costs, or N -way mirroring with their very low efficiency. See also the remarks in Section 5 on the LDPC and Wiencko codes.

3.1.2 Localization Effects

The design characteristics of constant parity in-degree and rotational symmetry are key features of the WEAVER codes. They enable the “localization” of many operations on the stripe. We have seen two examples of this: (a) in Section 2.1.1 we saw reconstruction requiring access to a small bounded (independent of stripe size) subset of the stripe; and (b) in Section 2.3.2 we saw write lock zones for two codes that are also small, bounded subsets of the stripe.

These two examples are typical of any WEAVER code (in fact, any code with parity in-degree bounded independent of the stripe size). The write lock zone (see Section 1.1) can be determined by examining the 2-neighborhood of the target element in the data/parity graph (see Figure 1 – the 2-neighborhood is the set of nodes in the graph within a distance two of the target element’s node). With t parities each having k out edges (one of which is the target element), this bounds the write lock zone to at most $t(k - 1) + t = tk$ data and parity elements (so at most tk devices as well – for some WEAVER codes, the actual number of devices is smaller). This is independent of the stripe size n , providing a proof of the localized write lock zone.

In contrast, even RAID5 has a write lock zone that is effectively the entire stripe, since the 2-neighborhood of an element is the entire remainder of the stripe. This is a consequence of the parity in-degree determined as a function of the stripe size.

Similar localization effects occur during rebuild. A

rebuild of one or more lost strips in a WEAVER code only requires access to a fixed and bounded set of strips. This set is at most the union of the 2-neighborhoods for all the data elements on all the lost strips (and is independent of n as well). Also, as we saw in Section 2.1.1, certain multi-strip failures may partition themselves into independent smaller failure scenarios. Reconstruction algorithms generally get more complicated with the number of inter-related failures so partitioning a multiple failure case into two or more independent failure cases can have a significant performance advantage and enable parallelism. For example, consider the recovery costs of EVENODD(p, n) [2] and WEAVER($n, 2, 2$) code when two strips fail. For $n \geq 6$, the EVENODD always requires accessing $n - 2$ devices, whereas most cases of WEAVER recovery involve only 4 devices, and the other cases only require 2 devices to be accessed!

3.1.3 Symmetry

The vertical layout of data and parity together with the symmetry properties (balanced parity in-degree and data out-degree and the rotational pattern) provide natural load balancing across all the devices. Multiple instances of the WEAVER codes can be stacked on the same set of devices with a simple host-to-strip addressing and also a simple logical strip-to-physical-device labeling. In contrast, the parity rotation of RAID5 vs RAID4 requires more complicated logical/physical addressing models.

Furthermore, multiple WEAVER code instances with different fault tolerances can be easily stacked on the same collection of devices (provided the number of devices is sufficiently large). This enables different reliability classes of logical volumes on the same set of devices. This is possible with other codes, but generally requires more complex logical/physical addressing.

3.1.4 Variability of Stripe Size and Fault Tolerance

The localization property mentioned above enables the WEAVER stripes to be expanded or shrunk with only local effects to the stripe; that is, not all devices need to be accessed and data or parity moved around. For example, in the WEAVER($n, 2, t$) codes, new devices/strips can be inserted into the stripe, and only the nearby devices need to have parity recomputed. See the additional comments in Section 5.

With the WEAVER(n, t, t) codes, it is further possible to change, *on-the-fly*, the fault tolerance of a single stripe in the system (either up or down) by simply recomputing the parity values. No remapping of either the host addressing or the strip labeling is needed. The only requirement is that the stripe size is supported for the higher fault tolerance. This enables more autonomic adaptability and is not possible with (almost) any other

code. In addition, by using a chain of $\kappa_1(0)$ subsets (e.g., those marked by an asterisk in Table 1), the recomputation step involves only adding a single new data value into each parity value and then storing the new parity values in the appropriate strips (which changes only if the offset changes). This is significantly more efficient than recomputing all the parity from scratch. Note that “adding a data value” can be used to either lower or raise the fault tolerance.

We believe that these features in particular make the WEAVER codes best suited for dRAID (distributed Redundant Arrangement of Independent Devices) systems involving network-connected storage nodes. Such systems will likely have data sets with varying reliability and performance requirements. Such sets may be distributed across different but intersecting sets of nodes. The WEAVER variability of stripe size and fault tolerance enable a dRAID data distribution algorithm to focus on user data layout (e.g., for load-balancing) and to achieve a balanced parity distribution as a natural consequence of the code itself. In addition, the chain of design sets for $\text{WEAVER}(n, t, t)$ codes allows the system to change fault tolerance with minimal network bandwidth utilization. Each node reads both its data and parity values, and sends only a single data value over the network, performs a single XOR operation, sends the recomputed parity value over the network and then performs a single disk write operation of the new parity). The parity “send” step is only required if the offset changes; more interestingly, the data “send” step may be skipped for some parity defining set subset chains. This operation is then both load-balanced and disk and network efficient.

3.1.5 Short Write IOs

For most of the $\text{WEAVER}(n, k, t)$ codes, the short write IO cost in device accesses (e.g., disk seeks) is equal to $2(t + 1)$. For the parity update algorithm that uses the parity delta, this seek cost is optimal for any t fault tolerant code. Many codes have even higher short write IO costs, when a given data element touches more than t parity elements (and strip sizes are large – see Section 3.1.7). For example, the EVENODD codes [2, 3] have this property for some elements.

Furthermore, only codes of efficiency (approximately) 50% can achieve better short write IO seek costs than the typical $2(t + 1)$. For example, a t -fault tolerant Reed-Solomon code can perform a short write in $2t$ seeks but only if $n = 2t$ (so efficiency 50%) or in $(2t + 1)$ seeks only if $n = 2t + 1$ (so efficiency close to 50%). In these cases, the stripe size is fixed as a function of the fault tolerance t . Mirroring achieves the best short write IO seek costs ($t + 1$) but also has the lowest efficiency.

In contrast, some WEAVER codes achieve better short write IO seek costs for a given fault toler-

ance and for *any* valid stripe size. For example, the $\text{WEAVER}(n, 2, 2)$ code (see Section 2.1.1) enables an implementation of a short write with 5 IOs (one less than is typical). This is achieved by reading the west and east neighbors of the target data element, computing the two new parities (from the parity equations) and writing the two new parities and one new data.

Similarly, by amortizing two neighbor data element reads into one longer read, the other two ad hoc WEAVER codes with parity in-degree equal to 2 can achieve a short write IO seek cost of 6 (for fault tolerance 3, Section 2.3.1) and 7 (for fault tolerance 4, Section 2.3.2). We emphasize that these IOs are not all of the same length, and a fairer comparison should take this into account (we do not do that here since seeks dominate device access costs; but see [9] for a more thorough analysis of these issues).

3.1.6 Multiple Strip Writes

Many of the WEAVER codes can amortize a significant number of device accesses required for consecutive multi-strip host writes. We explain this in detail for the $\text{WEAVER}(n, 2, 2)$ code (Section 2.1.1) and leave it to the reader to see how this principle can be applied in other WEAVER constructions. We will contrast the WEAVER behavior with EVENODD [2] which is also 2 fault tolerant, though this analysis applies to many other 2-fault tolerant codes, including Reed-Solomon [14].

Suppose the system gets a host write for m consecutive strips in a stripe of size n . For the EVENODD code, there are two efficient implementations. The first implementation reads all the m strips and the 2 parity strips, computes the new parity strips and writes the m data strips and 2 parity strips for a total of $2m + 4$ device accesses. The second implementation reads the $n - 2 - m$ other (2-neighborhood) data strips, computes the parity and writes $m + 2$ data and parity strips for a total of n device accesses. The optimum number of device accesses is then $\min(2m + 4, n)$.

In contrast, the following diagram shows how this could be implemented for a $\text{WEAVER}(n, 2, 2)$ code. In the diagram, an R indicates a data element that we read (only), a W indicates a data element that we write (this is a target of the host IO), P indicates a parity element that we write (we do not read any parity or old data).

	R	W	W	\dots	W	R
P	P	P	\dots	P		

For the same m consecutive multi-strip write, we read only two data elements (the two indicated by R on the west and east ends of the top row), and write $m - 1$ strips with *both* data and parity (as a single seek), one data element W (on the right side of the top row) and the two parity elements labeled P on the left, bottom row. This

totals $2 + m - 1 + 1 + 2 = m + 4$ device accesses and is better than $\min(2m + 4, n)$ for EVENODD provided $m \leq n - 4$. (For $m = n - 3, n - 2, n - 1, n$ the device accesses counts are $n + 1, n + 2, n + 1, n$, respectively, indicating a small disadvantage in these cases.)

3.1.7 Host IO Size Uniformity

XOR-based erasure codes have typically two alternative implementations with respect to size (number of bytes) that are mapped to each strip. One choice is to map a strip to a small unit of the device, e.g., 512B or perhaps 4KB. In this case, a host short write maps to a full strip write and the parity computations involve all (or most) of the parity elements. Generally, Reed-Solomon codes are implemented as extreme examples of this where a strip is a byte. XOR-based codes may be implemented in this way as well, but multiple rows impose fragmentation of the parity computations.

The alternative is to map a strip to a unit comparable to a moderate size host IO (say, 256KB as is typically done in RAID5). In these cases, elements are much larger units and a host short write affects only a subportion of an element. With this implementation, the host short write costs can scale to larger host IO lengths, up to the size of the element, incurring additional costs that are only linear functions of the length. There are no additional seeks or other computational costs such as additional XOR formulas.

For a fixed strip size, more data rows in the code imply smaller element size (for a fixed strip size), and hence limitations on the advantages of this uniform and linear scaling. In this regard, (most of) the WEAVER codes are optimal because they have only one data row (the ad hoc constructions have two data rows, so are near optimal). They also do not suffer from excessive XOR fragmentation for the same reason. (Clearly, RAID5 and N -way mirroring have these properties as well but they are at opposite ends of the fault-tolerance/efficiency spectrum, with WEAVER codes occupying the middle ground. See Section 5 for additional comments.)

3.2 Disadvantages – Efficiency

The primary disadvantage of the WEAVER codes is their limited efficiency (at most 50%). On the other hand, WEAVER codes are optimally efficient among all possible codes of fault tolerance t and parity in-degree $k \leq t$ as the following argument shows. Suppose an erasure code with fault tolerance t has N data elements and Q parity elements and that each parity element has parity in-degree bounded above by k . Each data element must touch at least t parity elements. Counting edges in the parity graph (as in the WEAVER example in Figure 1), we see that the number of edges E is at least Nt (counting the edges coming out of the data nodes) and

at most Qk (counting the edges coming into the parity nodes); that is, $Nt \leq E \leq Qk$. The efficiency is

$$\frac{N}{N+Q} = \frac{Nt}{Nt+Qt} \leq \frac{Qk}{Qk+Qt} = \frac{k}{k+t},$$

with equality in the case of the WEAVER codes. In addition, we clearly see the trade-off of efficiency for simplicity, fault tolerance and the other positive features of these codes. (Other codes, including RAID5 have performance/efficiency trade-offs, but those trade-offs are as functions of the stripe size with constant fault tolerance – for WEAVER codes, it is a function of the fault tolerance regardless of the stripe size.)

4 Testing methodology

We have mentioned above that we search parameter spaces for valid configurations, that is, configurations that provide the requisite fault tolerance. Our methodology is the following (see also [10]). For each fault tolerance t and each choice of parity defining set (including offset), and for each stripe size n , we construct the binary generator matrix for the WEAVER code of those parameters. This matrix has rows indexed by the data elements and columns indexed by the data and parity elements. A column with a single one indicates a data element; a column with at least 2 ones indicates a parity element (and the formula used to compute it – that is, the data elements touching it). We view the generator matrix in column block form where each block corresponds to a strip; the columns in a block correspond to data and parity elements on the same strip. The blocks are indexed by the numbers $\{0, 1, \dots, n-1\}$ since there are n strips. The generator matrix then maps the input user data into data and parity elements, and the block structure shows how they are organized on strips.

We can simulate strip loss by removing (or zeroing) the columns of the corresponding block. Consequently, to test this code for the required fault tolerance t , we execute the following pseudo-code:

1. For each t -sized subset $T \subseteq \{0, 1, \dots, n-1\}$:
 - (a) Remove the column blocks from the generator matrix indexed by the elements of T (simulate strip loss).
 - (b) If the binary row rank of the reduced matrix equals the number of rows, continue; else return “Invalid”.
2. Return “Valid”

The algorithm returns “Valid” if and only if the given generator matrix defines a t fault tolerant code. Each reduced matrix represents a set of equations that maps the “unknown” data values into the “known” data and parity values (those that are not lost). The row rank of the reduced matrix equals the number of rows if and only if

this system of equations is solvable; that is, if and only if all the “unknown” data values can be reconstructed from the “known” data and parity values. This test succeeds for all specific failure cases T if and only if the code is t fault tolerant.

For rotationally symmetric codes like the WEAVER codes, one can restrict the search space a bit by requiring that $0 \in T$. Other such restrictions may be used to reduce the search space further (e.g., see [6]).

As n and t grow, this search space grows rather rapidly, as there are $\binom{n-1}{t-1}$ such cases to consider (and the matrices get larger as $O(n^2)$). Other optimizations are possible that reduce these costs.

5 Related Work – Other codes

The WEAVER codes can be compared to any other erasure code suitable for storage systems (we have mentioned some already such as Reed-Solomon and EVEN-ODD). For ease of comparison, we divide the set of known erasure codes into different categories and give (non-exhaustive) examples in each category. In a category by themselves are the Reed-Solomon codes [14], which are MDS but require complex finite field arithmetic. Second are XOR-based codes that are MDS. These come in two types: vertical codes such as the X-code [17], BCP [1] or ZZS codes [18] and horizontal codes such as EVENODD [2, 3], Blaum-Roth [4], or Row-Diagonal Parity codes [5]. Finally, there are non-MDS codes that are XOR-based. These subdivide into three categories based on efficiency: N -way mirroring (trivially XOR-based) with efficiency less than 50%, the Gibson, *et al*, codes [8] with efficiency larger than 50%, and two codes with efficiency 50% exactly. In the last category, we have the LSI code [16] (in fact a subcode of one of the Gibson *et al* codes) and one 3 fault tolerant Blaum-Roth binary code [4]. In the second category, we also have the LDPC codes (see [7, 11, 13, 12]) and the Wiencko codes [15, 6] which we address separately.

LDPC codes [7, 11] were originally designed for communication channels but have recently been studied in the context of storage applications over wide-area networks [13, 12]. In these applications, random packet loss (or delay) is the dominant erasure model (not total device failure), so a typical “read” *a priori* assumes random erasures and hence is always in reconstruction mode. Because of this, good LDPC codes have highly irregular graph structures but can have high fault tolerance and near optimal efficiency (both in the sense of expected value, however). In contrast, the WEAVER codes are designed for a traditional storage model where reads involve reconstruction algorithms *only if* the direct read of the user data fails. In addition, WEAVER codes have very regular graphs, and relatively high fault tolerance over a wide range of stripe sizes.

The Wiencko codes are presented in the patent [15] and patent application [6] as methodologies for formulating and testing a specific code instance via design patterns. This is similar to the BCP [1] patent. The Wiencko codes are vertical codes (with layout as in Figure 2) and can utilize rotational symmetry. Valid constructions meet certain parameter constraints, and include MDS design possibilities. Few examples are given in these references, however. The construction methodology differs from WEAVER codes in that no *a priori* restrictions are placed on the parity in-degree. In fact, some examples have differing degree for different parity values; so are less regular and uniform than required in WEAVER codes. Essentially any vertical code such as the X-code [17] and BCP [1] can also be (re)constructed by a Wiencko formulation.

With the exception of the Reed-Solomon codes, N -way mirroring, LDPC and Wiencko codes, none of the codes have exceptionally high fault tolerance. There are variations of EVENODD [3] that are ≥ 3 fault tolerant; there is one instance of a BCP code of three fault tolerance on 12 strips; the Blaum-Roth [4] binary code of efficiency 50% and two codes in [8] are 3 fault tolerant. As far as we know, none of the other codes have variants that can tolerate more than 2 failures. The WEAVER codes can have very high fault tolerance (up to 12 and perhaps beyond). Compared to Reed-Solomon codes, they are significantly simpler but less efficient. Compared to N -way mirroring, they are more complex but more efficient. The WEAVER codes provide alternative interpolating design points between Reed-Solomon and N -way mirroring over a long range of fault tolerances.

As we mentioned in Section 3.1.5, only codes with efficiency approximately 50% can implement a host short write with fewer IO seeks than $2(t+1)$; the implementation in fact must compute parity from new data and the 2-neighborhood dependent data. To achieve IO seeks costs less than $2(t+1)$, this 2-neighborhood must be small. Special subcodes of the horizontal codes (both MDS and non-MDS) can achieve this but only if the stripe size is bounded as a function of the fault tolerance: $n = 2t$ or $n = 2t + 1$. The Blaum-Roth [4] three fault tolerant (binary) code is equivalent to a very special case of Reed-Solomon with $t = 3$ and so can be implemented with a 6 IO seeks (at efficiency 50%). In these implementations, the strip size must be comparable to the short write IO size (see Section 3.1.7) so that a short write contains a strip. Only the LSI code [16] and the WEAVER($n, 2, t$) codes support variable stripe sizes of fixed fault tolerance and improved short write IO seek costs. These can both be implemented with large single element strips gaining the advantages of host IO size uniformity over a longer range of sizes (see Section 3.1.7).

All the MDS codes have the property that parity in-

degree increases with stripe size, for constant t . Consequently, the advantages of WEAVER codes that result from bounded parity in-degree (see Section 3.1.2) can not be achieved with MDS codes of similar fault tolerance. Here again is a performance/efficiency trade-off.

All the 2 fault tolerant XOR-based MDS codes (and some 3 fault tolerant codes as well) share the property that the number of elements per row (row count) increases with increasing stripe size. For example, for EVENODD, the row count is $p - 1$ where $p \geq n + 2$ and p is prime; for the X-code, the row count equals the stripe size (and must be a prime as well). This has consequences for stripe size flexibility. For horizontal codes such as EVENODD, Row-Diagonal Parity, or Blaum-Roth, flexibility in stripe size can be attained either by selecting a large row count to start with, or by changing the row count with stripe size. The latter is prohibitively complex in practice. The former, initial large row count, increases the fragmentation and XOR-complexity of parity computations. For example, the scalability of host IO size (see Section 3.1.7) rapidly degrades with increasing row count. For the vertical codes, changing stripe sizes implies changing row counts and that is prohibitive for on-the-fly changes (this may not be true for certain Wiencko codes, though this has not been established). In contrast, the WEAVER codes maintain constant XOR complexity with changes in stripe size (XOR complexity only increases as fault tolerance increases, which is a necessary effect).

The Gibson *et al* codes (of efficiency greater than 50%) share a number of the good qualities of the WEAVER codes, including the host IO size uniformity (because they have only one row). They are, however, horizontal codes and so require parity rotation for load balancing, only tolerate at most 3 failures and have large minimum stripe sizes. Furthermore, to maintain balance and symmetry, they must restrict stripe sizes to specific values. We believe these codes are reasonable choices for performance/efficiency trade-offs for 2 or 3 fault tolerant codes if efficiency beyond 50% is required. As we have seen, though, the WEAVER codes have a number of additional advantages, a greater range of fault tolerance and better natural balance and symmetry.

The LSI code is very similar to the special case WEAVER($n, 2, 2$) code detailed in Section 2.1.1, and as mentioned is a subcode of a Gibson *et al* code. Each parity value is computed from two data elements, but instead of being placed below in a new row (and new strip), each parity value is placed on a new device separate from any data in the stripe (so it is a horizontal code). Besides being limited to tolerating only 2 failures, the specific layout of the LSI code implies two restrictions on stripe size: $n \geq 6$ and n must be even. The WEAVER($n, 2, 2$) code requires only $n \geq 4$ and has no such even/odd re-

striction. In addition, the WEAVER vertical code layout again provides natural load balancing under mixed read/write host IOs without any special parity rotation as would be required for the LSI code.

6 Open Problems

There are still a number of open questions and missing constructions. We list a few here:

- Find constructions of WEAVER codes where k divides t , $4 \leq k < t \leq 8$ (see Section 2.2).
- For $k = t$ (Section 2.1), determine the minimum valid stripe size and the parity defining sets that achieve this minimum. More generally, resolve the same issue for any t and $k \leq t$.
- For a given parity defining set, determine the stripe size n_0 so that all stripe sizes $n \geq n_0$ are valid (see the next theorem); more generally determine the complete set of valid stripe sizes.

Theorem 1. *Let a WEAVER(n, k, t) be constructed from a collection of parity defining sets $\{\kappa(j) : 0 \leq j \leq q - 1\}$ (one for each parity row). Let w be the largest element in these sets. Then the fault tolerance of the code is constant for all $n > tw$.*

Proof. (Idea) The fault tolerance should not change once n is larger than the largest window of devices that are (a) touched by some data element and its 2-neighborhood and (b) can be affected by t failures. This window defines the localization of the code (write lock zone and rebuild zones) and is dependent only on w, k, t . For a given data element, the parity it touches are within a neighborhood of at most w strips. There are at most t such neighborhoods that can be affected. Consequently, once $n > tw$, the failures are localized to a zone independent of n and the result follows. Note, we claim only constant, but not necessarily t , fault tolerance. \square

7 Summary

In this paper, we introduced the WEAVER codes, families of XOR-based erasure codes suitable for storage systems (either RAID arrays or dRAID node-based systems). These codes have a number of significant features: (a) they are designed with simplicity and symmetry for easy implementation; (b) they have constrained parity in-degree for improved computational performance; (c) they are vertical codes for inherent load-balance; (d) they have constructions with very high fault tolerance; (e) they support all stripe sizes above some minimum (determined as a function of each specific construction, but generally dependent on the fault tolerance); (f) there are families with efficiency equal to 50% as well as families of lower efficiency (independent of fault tolerance and stripe size). The WEAVER codes

provide system designers with great flexibility for fault tolerance and performance trade-offs versus previously published codes. They provide a middle ground between the performance advantages but low efficiency of N -way mirroring and the lower performance but higher efficiency of codes such as Reed-Solomon. All these features make the WEAVER codes suitable for any storage system with high fault tolerance and performance requirements; they are perhaps best suited to dRAID systems where flexibility in stripe sizes, fault tolerance and autonomic considerations drive design choices.

8 Acknowledgements

The author extends his thanks and appreciation to Jeff Hartline, Tapas Kanungo and KK Rao. Jeff, in particular, showed the author the relationship between parity in-degree and efficiency, thereby indirectly offering the challenge to construct optimal codes under these constraints. We also want to thank the Blue Gene/L support team at IBM's Almaden Research Center for the opportunity to run many of the larger experiments on their system (and their assistance). Testing stripe sizes in 40-60 ranges, with fault tolerance 10 and tens of thousands of parity defining sets (as we did for the results in Table 1) required considerable computing power. Finally, we thank the reviewers for reminding us about LDPC codes and for pointing us to the Wiencko codes.

References

- [1] S. Baylor, P. Corbett, and C. Park. Efficient method for providing fault tolerance against double device failures in multiple device systems, January 1999. U. S. Patent 5,862,158.
- [2] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44:192–202, 1995.
- [3] M. Blaum, J. Brady, J. Bruck, J. Menon, and A. Vardy. The EVENODD code and its generalization. In J. Jin, T. Cortest, and R. Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 14, pages 187–208. IEEE and Wiley Press, New York, 2001.
- [4] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45:46–59, 1999.
- [5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.
- [6] L. J. Dickson. Data redundancy methods and apparatus, October 2003. U. S. Patent Application US2003/0196023 A1.
- [7] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, Cambridge, MA, 1963.
- [8] G. A. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz, and D. A. Patterson. Failure correction techniques for large disk arrays. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 123–132, Boston, MA, 1989.
- [9] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and KK Rao. Performance metrics for erasure codes in storage systems. Technical Report RJ 10321, IBM Research, San Jose, CA, 2004.
- [10] J. L. Hafner, V. Deenadhayalan, KK Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies*, San Francisco, CA USA, December 2005.
- [11] M. G. Luby, M. Mitzenmacher, A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47:569–584, 2001.
- [12] J. S. Plank, R. L. Collins, A. L. Buchsbaum, and M. G. Thomason. Small parity-check erasure codes – exploration and observations. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, 2005.
- [13] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, 2004.
- [14] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8:300–304, 1960.
- [15] J. A. Wiencko, Jr., K. Land, and L. J. Dickson. Data redundancy methods and apparatus, April 2003. U. S. Patent 6,557,123 B1.
- [16] A. Wilner. Multiple drive failure tolerant raid system, December 2001. U. S. Patent 6,327,672 B1.
- [17] L. Xu and J. Bruck. X-code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, IT-45:272–276, 1999.
- [18] G. V. Zaitsev, V. A. Zinovev, and N. V. Semakov. Minimum-check-density codes for correcting bytes of errors. *Problems in Information Transmission*, 19:29–37, 1983.

On multidimensional data and modern disks

Steven W. Schlosser[†], Jiri Schindler[‡], Stratos Papadomanolakis^{*}, Minglong Shao^{*},
Anastassia Ailamaki^{*}, Christos Faloutsos^{*}, Gregory R. Ganger^{*}
[†]Intel Research Pittsburgh, [‡]EMC Corporation, ^{*}Carnegie Mellon University

Abstract

With the deeply-ingrained notion that disks can efficiently access only one dimensional data, current approaches for mapping multidimensional data to disk blocks either allow efficient accesses in only one dimension, trading off the efficiency of accesses in other dimensions, or equally penalize access to all dimensions. Yet, existing technology and functions readily available inside disk firmware can identify non-contiguous logical blocks that preserve spatial locality of multidimensional datasets. These blocks, which span on the order of a hundred adjacent tracks, can be accessed with minimal positioning cost. This paper details these technologies, analyzes their trends, and shows how they can be exposed to applications while maintaining existing abstractions. The described approach can achieve the best possible access efficiency afforded by the disk technologies: sequential access along primary dimension and access with minimal positioning cost for all other dimensions. Experimental evaluation of a prototype implementation demonstrates a reduction of overall I/O time for multidimensional data queries between 30% and 50% when compared to existing approaches.

1 Introduction

Large, multidimensional datasets are becoming more prevalent in both scientific and business computing. Applications, such as earthquake simulation and oil and gas exploration, utilize large three-dimensional datasets representing the composition of the earth. Simulation and visualization transform these datasets into four dimensions, adding time as a component of the data. Conventional two-dimensional relational databases can be represented as multidimensional data using online analytical processing (OLAP) techniques, allowing complex queries for data mining. Queries on this data are often ad-hoc, making it difficult to optimize for a particular workload or access pattern. As these datasets grow in size and popularity, the performance of the applications that access them growing in importance.

Unfortunately, storage performance for this type of data is often inadequate, largely due to the one-dimensional abstraction of disk drives and disk arrays. Today's data placement techniques are commonly predicated on the assumption that multidimensional data must

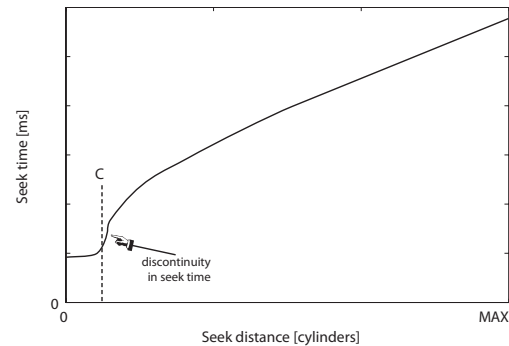


Figure 1: **Notional seek curve of a modern disk drive.** The seek time profile of a modern disk drive consists of three distinct regions. For cylinder distances less than C , the seek time is constant, followed by a discontinuity. After this point of discontinuity, the seek time is approximately the square root of the seek distance. For distances larger than one third of the full seek distance, the seek time is a linear function of seek distance. To illustrate the trend more clearly, the X axis is not drawn to scale.

be serialized when stored on disk. Put another way, the assumption is that spatial locality cannot be preserved along all dimensions of the dataset once it is stored on disk. Various data placement and indexing techniques have been proposed over the years to optimize access performance for various data types and query workloads, but none solve the fundamental problem of preserving locality of multidimensional data.

Some recent work has begun to chip away at this assumption [13, 27], showing that locality in two-dimensional relational databases can be preserved on disk drives, but we believe that these studies have only scratched the surface of what is possible given the characteristics and trends of modern disks. In this paper, we show that modern disk drives *can* physically preserve spatial locality for multidimensional data. Our technique takes advantage of the dramatically higher densities of modern disks, which have increased the number of tracks that can be accessed within the time that it takes the disk head to settle on a destination track. Any of the tracks that can be reached within the settle time can be accessed for approximately equal cost, which contrasts with the standard “rule of thumb” of disk drive technology that longer seek distances correspond to longer seek times.

Figure 1 illustrates the basic concept using a canonical seek curve of a modern disk drive. In contrast to conven-

tional wisdom, seek time for small distances (i.e., fewer than C cylinders, as illustrated in the figure) is often a constant time equal to the time for the disk head to settle on the destination cylinder. We have found that C is not trivially small, but can be as high as 100 cylinders in modern disks. This means that on the order of 100 disk blocks can be accessed for equal cost from a given starting block. We refer to these blocks as being *adjacent* to the starting block, meaning that any of them can be accessed for equal cost.

In this paper, we explain the adjacency mechanism, detailing the parameter trends that enable it today and will continue to enable it into the future. We describe the design and implementation of a prototype disk array logical volume manager that allows applications to identify and access adjacent disk blocks, while hiding extraneous disk-specific details so as to not burden the programmer. As an example, we also evaluate a data placement technique that maps a three- and four-dimensional dataset onto the logical volume, preserving physical locality directly on disk, and improving spatial query performance by between 30% and 50% over existing data placements.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 describes details of the adjacency mechanism, how it can be implemented in modern disks, and historic and projected disk parameter trends that enable the adjacency mechanism. Section 4 analyzes data obtained from measurements of several state-of-the-art enterprise-class SCSI disks to show how their characteristics affect the properties of the adjacency mechanism. Section 5 shows how adjacency can be expressed to applications without burdening them with disk-specific parameters. Section 6 evaluates the efficiency of adjacent access on a prototype system using microbenchmarks as well as 3D and 4D spatial queries.

2 Background and related work

Effective multidimensional data layout techniques are crucial for the performance of a wide range of scientific and commercial applications. We now describe the applications that will benefit from our approach and show that existing techniques do not address the problem of preserving the locality of multidimensional data accesses.

2.1 Multidimensional datasets

Advances in computer hardware and instrumentation allow high-resolution experiments and simulations that improve our understanding of complex physical phenomena, from high-energy particle interactions to combustion and earthquake propagation. The datasets involved in modern scientific practice are massive and multidimensional. Modern simulations produce data at the staggering rate of multiple terabytes per day [21],

while high energy collision experiments at CERN are expected to generate raw data of a petabyte scale [33]. Realizing the big benefits of the emerging data-driven scientific paradigm heavily depends on our ability to efficiently process these large-scale datasets.

Simulation applications are a great example for the storage and data management challenges posed by large-scale scientific datasets. Earthquake simulations [2] compute the propagation of an earthquake wave over time, given the geological properties of a ground region and the initial conditions. The problem is discretized by sampling the ground at a collection of points and the earthquake's duration as a set of time-steps. The simulator then computes physical parameters (like ground velocity) for each discrete ground point and for each time step. Post-processing and visualization applications extract useful information from the output.

The difficulties in efficiently processing simulation output datasets lie in their volume and their multidimensional nature. Storing one time-step of output requires many gigabytes, while a typical simulation generates about 25,000 such time-steps [40]. An earthquake simulation dataset is four-dimensional: it encodes three-dimensional information (the 3D coordinates of the sample points) at each time-step. Post-processing or visualization applications query the output, selecting the simulation results that correspond to ranges of the 4D coordinate space. As an example of such *range queries*, consider a "space-varying" query that retrieves the simulated values for all the ground points falling within a given 3D region for only a single time-step. Similarly, "time-varying" queries generate waveforms by querying the simulated values for a single point, but for a range of time-steps.

Unfortunately, naive data layout schemes lead to sub-optimal I/O performance. Optimizing for a given class of queries (e.g., the 3D spatial ranges), results in random accesses along the other dimensions (e.g., the time dimension). Due to the absence of an appropriate disk layout scheme, I/O performance is the bottleneck in earthquake simulation applications [40].

Organizing multidimensional data for efficient accesses is a core problem for several other scientific applications. High energy physics experiments will produce petabyte-scale datasets with hundreds of dimensions [33]. Astronomy databases like the Sloan Digital Sky Survey [15] record astronomical objects using several other attributes besides their coordinates (brightness in various wavelengths, redshifts, etc.). The data layout problem becomes more complex with an increasing number of dimensions because there are more query classes to be accommodated.

In addition to data-intensive science applications, large-scale multidimensional datasets are typically used

in On-Line Analytical Processing (OLAP) settings [14, 34, 39]. OLAP applications perform complex queries on large volumes of financial transactions data in order to identify sales trends and support decision-making. OLAP datasets have large numbers of dimensions, corresponding, for example, to product and customer characteristics, and to the time and geographic location of a sale. Performance of complex multidimensional analysis queries is critical for the success of OLAP and a large number of techniques have been proposed for organizing and indexing multidimensional OLAP data [7, 18, 32, 41].

2.2 Limitations of conventional placement

Efficient multidimensional data access relies on maintaining locality so that “neighboring” objects in the multidimensional (logical) space are stored in “neighboring” disk locations. Existing multidimensional layout techniques are based on the standard linear disk abstraction. Therefore, to take advantage of the efficient sequential disk access, neighboring objects in the multidimensional space must be stored in disk blocks with similar logical block numbers (*LBNs*). Space-filling curves, such as the Hilbert curve [17], Z-ordering [22] and Gray-coding [10] are mathematical constructs that map multidimensional points to a 1D (linear) space, so that nearby objects in the logical space are as close in the linear ordering as possible.

Data placement techniques that use space-filling curves rely on a simplified linear disk model, ignoring low-level details of disk performance. The resulting linear mapping schemes break sequential disk access, which can no longer be used for scans along any dimension, only to ensure that range queries do not result in completely random I/O. Furthermore, as analysis [20] and our experiments suggest, the ability of space-filling curves to keep neighbors in any dimensions physically close on the disk deteriorates rapidly as the number of dimensions increases. Our work revisits the simplistic disk model and removes the need for linear mappings. The resulting layout schemes maintain sequential disk bandwidth, while providing efficient access along any dimension, even for datasets with large dimensionality.

Besides space-filling curves, other approaches rely on parallel I/O to multiple disks. Declustering schemes [1, 4, 5, 11, 19, 23] partition the logical multidimensional space across multiple disks, so that range queries can take advantage of the aggregate bandwidth available.

2.3 Limitations of indexing

The need to efficiently support multidimensional queries has led to a large body of work on indexing. Multidimensional indexes like the R-tree [16] and its variants are disk-resident data structures that provide fast access

to the data objects satisfying a range query. With an appropriate index, query processing requires only a fraction of disk block accesses, compared to the alternative of exhaustively searching the entire dataset. The focus of multidimensional indexing research is on minimizing the number of disk pages required for answering a given class of queries [12].

Our work on disk layout for multidimensional datasets differs from indexing. It improves the performance of *retrieving* the data objects that match a given input query and not the efficiency of *identifying* those objects. For example, a range query on a dataset supported by an R-tree can result in a large number of data objects that must be retrieved. Without an appropriate data layout scheme, the data objects are likely to reside in separate pages at random disk locations. After using the index to identify the data objects, fetching them from the disk will have sub-optimal, random access performance. Multidimensional indexing techniques are independent of the underlying data layout and do not address the problem of maintaining access locality.

2.4 Storage-oriented approaches

Recently, researchers have focused on the lower level of the storage system in an attempt to improve performance of multidimensional queries [13, 27, 28, 38]. Part of the work revisits the simple disk abstraction and proposes to expand the storage interfaces so that the applications can be more intelligent. Schindler et al. [26] explored aligning accesses to disk drive track-boundaries to get rid of rotational latency. Gorbatenko et al. [13] and Schindler et al. [27] proposed a secondary dimension on disks, which they utilized to create a more flexible database page layout [30] for two-dimensional tables. Others have studied the opportunities of building two dimensional structures to support database applications with new alternative devices, such as MEMS-based storage devices [28, 38]. The work described in this paper challenges the underlying assumption of much of this previous work by showing that the characteristics of modern disk drives allow for efficient access to multiple dimensions, rather than just one or two.

3 Multidimensional disk access

The established view is that disk drives can efficiently access only one-dimensional data mapped to sequential logical blocks. This notion is further reinforced by the linear abstraction of disk drives as a sequence of fixed-size blocks. Behind this interface, disks use various techniques to further optimize for such sequential accesses. However, modern disk drives can allow for efficient access in more than one dimension. This fundamental change is based on two observations of technological trends in modern disks:

1. Short seeks of up to some cylinder distance, C , are dominated by the time the head needs to settle on a new track;
2. Firmware features internal to the disk can identify and thus access blocks that require no rotational latency after a seek.

By combining these two observations, it is possible to construct access patterns for efficient access to multidimensional data sets despite the current linear abstractions of storage systems.

This section describes the technical underpinnings of the mechanism that we exploit to preserve locality of multidimensional data on disks. We begin by describing some background of the mechanical operation of disks. We then analyze the technology trends of the relevant drive parameters, showing how the mechanism is enabled today and will continue to be enabled in future disks. Lastly, we combine the two to show how the mechanism itself works.

3.1 Disk background

Positioning. To service a request for data access, the disk must position the read/write head to the physical location where the data resides. First, it must move a set of arms to the desired cylinder, in a motion called seeking. Once the set of arms, each equipped with a distinct head for each surface, is positioned near the desired track, the head has to settle in the center of the track. After the head is settled, the disk has to wait for the desired sector to rotate underneath the stationary head before accessing it. Thus, the total positioning time is the sum of the seek time, settle time, and rotational latency components.

The dominant component of the total positioning time depends on the access pattern (i.e., the location of the previous request with respect to the next one). If these requests are to two neighboring sectors on the same track, no positioning overhead is incurred in servicing the second one. This is referred to as sequential access. When two requests are located on two adjoining tracks, the disk may incur settle time and some rotational latency. Finally, if the two requests are located at non-adjoining tracks, the disk may incur a seek, settle time, and possibly some rotational latency.

There are two possible definitions of adjoining tracks. They can be either two tracks with the same radius on different surfaces, or two neighboring tracks with different radii on the same surface. In the first case, different heads must be used to access the two tracks; in the second case, the same head is used for both. In both cases, the disk will have to settle the head above the correct track. In the former case, the settle time is incurred because (i) the two tracks on the two surfaces may not be perfectly aligned or round (called run out), (ii) the individual heads may not be perfectly stacked,

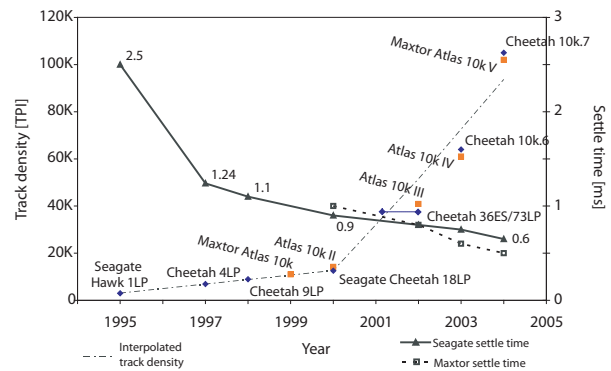


Figure 2: **Disk trends for 10,000 RPM disks.** Notice the dramatic increase in track density, measured in Tracks Per Inch (TPI), since 2000, while in the same period, head switch/settle time has improved only marginally.

and/or (iii) the arms may not be stationary (as any non-rigid body, they oscillate slightly). The former case is referred to as head switch between tracks of the same cylinder, while the latter is called a one cylinder seek.

Request scheduling. Disk drives use variants of shortest positioning time first (SPTF) request schedulers [29, 36], which determine the optimal order in which outstanding requests should be serviced by minimizing the sum of seek/settle time and rotational latency. To calculate the positioning cost, a scheduler must first determine the physical locations (i.e., (cylinder, head, sector offset)) of each request. It then uses seek time estimators encoded in the firmware routines to calculate the seek time and calculates residual rotational latency after a seek based on the offset of the two requests.

Layout. Disks map sequential *LBNs* to adjoining sectors on the same track. When these sectors are exhausted, the next *LBN* is mapped to a specific sector on the adjoining track to minimize the positioning cost (i.e., head switch or seek to the next cylinder). Hence, there is some rotational offset between the last *LBN* on one track and the next *LBN* on the next track. Depending on which adjoining track is chosen for the next *LBN*, this offset is referred to as track skew or cylinder skew.

3.2 Disk technology trends

The key to our method of enabling multidimensional access on disks is the relationship between two technology trends over the last decade: (1) the time for the disk head to settle at the end of a seek has remained largely constant, and (2) track density has increased dramatically. Figure 2 shows a graph of both trends for two families of (mostly) 10,000 RPM enterprise-class disks from two vendors, Seagate and Maxtor.

The growth of track density, measured in tracks per inch (TPI), has been one of the strongest trends in disk drive technology. Over the past decade, while settle time has decreased only by a factor of 5 [3], track densities

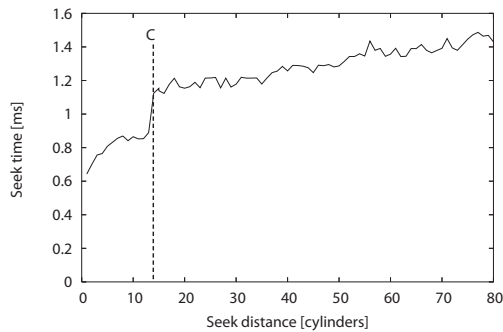


Figure 3: Atlas 10k III seek curve. First 80 cylinders.

experienced 35 fold increase, as shown in Figure 2. As track density continues to grow while settle time improves very little, more cylinders can be accessed in a given amount of time. With all else being equal, more data in the same physical area can be accessed in the same amount of time. However, increasing track density negatively impacts the settle time. With larger TPI, higher-precision controls are required, diminishing the improvements in settle time due to other factors. There are other factors that improve seek time such as using smaller platters [3], but these affect long and full-stroke seeks, whereas we focus on short-distance seeks.

In the past, the seek time for short distances between one and, say, L cylinders was approximately the square root of the seek distance [24]. However, the technology trends illustrated in Figure 2 lead to seek times that are nearly constant for distances of up to C cylinders, and which then increase as before for distances above C cylinders, as illustrated in Figure 1. Seeks of up to C cylinders are dominated by the time it takes a disk head to settle on the new track. These properties are confirmed by looking at the seek curve measured from a real disk, shown in Figure 3. The graph shows the seek curve for a Maxtor Atlas 10k III, a 10,000 RPM disk introduced in 2002. For this disk, $C = 12$ and settle time is around 0.8 ms. For clarity, the graph shows seek time for distances up to 80 cylinders, even though it has over 31,000 cylinders in total (see Table 1).

While settle time has always been a factor in positioning disk heads, the dramatic increase in bit density over the last decade has brought it to the fore, as shown in Figure 2. At lower track densities (i.e., for disks introduced before 2000), only a single cylinder can be reached within a constant settle time. However, with the large increase in TPI since 2000, up to C can now be reached. Section 4.3 examines seek curves for more disks.

The increasing track density also influences how data is laid out on the disk. While in the past, head switches would be typically faster than cylinder switches, it is the other way around for today's disks. With increasing TPI,

settling on the correct track with a different head/arm takes more time than simply settling on the adjoining track with the same head/arm assembly.

Disks used to lay out data first across all tracks of the same cylinder before moving to the next one, whereas most recent disks “stay” on the same surface for a number of cylinders, say C_{layout} , and move inward before switching to the next surface and going back. This mapping, which we term *surface serpentine*, also leverages the fact that seeks of up to C cylinders take a (nearly) constant amount of time. Put differently, the choice of C_{layout} must ensure that sequential accesses are still efficient even when two consecutive LBNs are mapped to tracks C_{layout} cylinders away. Figure 4 depicts the different approaches to mapping LBNs onto disk tracks.

3.3 Adjacent disk blocks

The combination of rapidly increasing track densities and slowly decreasing settle time leads to the seek curves shown above in which one of C neighboring cylinders can be accessed from a given starting point for equal cost. Each of these cylinders is composed of R tracks, and so, by extension, there are $d = R \times C$ tracks that can be accessed from that starting point for equal cost. The values of C and d are related very simply, but we differentiate them to illuminate a subtle, but important, detail.

The value of C is a measure of *how far* the disk head can move (in cylinders) within the settle period, while the value of d is used to enumerate the number of adjacent blocks that can be accessed within those cylinders. While each of these d tracks contain many disk blocks, there is one block on each track that can be accessed immediately after the head settles on the destination track, with no additional rotational latency. We identify these blocks as being *adjacent* to the starting block.

Figure 5 shows a drawing of the layout of adjacent blocks on disk. For a given starting block, there are d adjacent disk blocks, one in each of the d adjacent tracks. For simplicity, we show a disk with only one surface, so, in this case, R is one, and d equals C . During the settle time, the disk rotates by a fixed number of degrees, W , determined by the ratio of the settle time to the rotational period of the disk. For example, with settle time of 1 ms and the rotational period of 6 ms (i.e., for a 10,000 RPM disk), $W = 60^\circ$. Therefore, all adjacent blocks have the same angular (physical) offset from the starting block.

As settle time is not entirely deterministic (i.e., due to external vibrations or thermal expansion), it is useful to add some extra conservatism to W to avoid rotational misses, which lead to long delays. Adding conservatism to the value of W increases the number of tracks, d , that can be accessed within the settle time at the cost of added rotational latency. In practice, disks also add some conservatism to the best-case settle time when determining

	<i>Model</i>	<i>Year</i>	<i>TPI</i>	<i>Cylinders</i>	<i>Surfaces</i>	<i>Max. Cap.</i>	<i>1-cyl Seek</i>	<i>Full Seek</i>
Maxtor	Atlas 10k II	2000	14200	17337	20	73 GB	1 ms	12.0 ms
	Atlas 10k III	2002	40000	31002	8	73 GB	0.8 ms	11.0 ms
	Atlas 10k IV	2003	61000	49070	8	147 GB	0.6 ms	12.0 ms
	Atlas 10k V	2004	102000	81782	8	300 GB	0.5 ms	12.0 ms
	Atlas 15k II	2004	<i>unknown</i>	48242	8	147 GB	0.5 ms	8.0 ms
Seagate	Cheetah 4LP	1997	6932	6526	8	4.5 GB	1.24 ms	19.2 ms
	Cheetah 36ES	2001	38000	26302	4	36 GB	0.9 ms	11.0 ms
	Cheetah 73LP	2002	38000	29549	8	73 GB	0.8 ms	9.8 ms
	Cheetah 10k.6	2003	64000	49855	8	147 GB	0.75 ms	10.0 ms
	Cheetah 10k.7	2004	105000	90774	8	300 GB	0.65 ms	10.7 ms
	Cheetah 15k.4	2004	85000	50864	8	147 GB	0.45 ms	7.9 ms

Table 1: **Disk characteristics.** Data taken from manufacturers' specification sheets. The listed seek times are for writes.

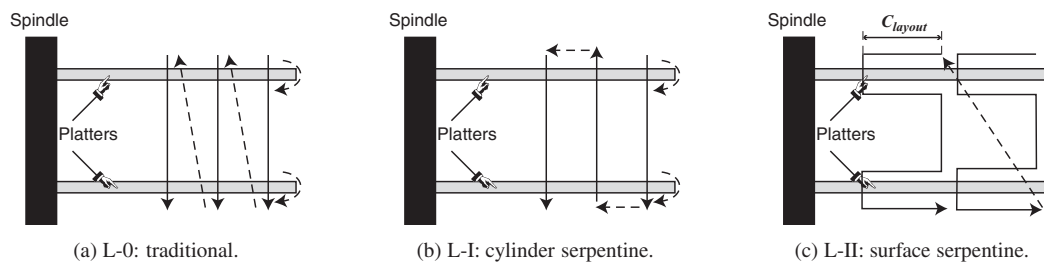


Figure 4: **Layout mappings adopted by various disk drives.**

cylinder and track skews for mapping *LBNs* to physical locations. For our example Atlas 10k III disk, the cylinder and track skews are 61° and 68° , respectively, or 1.031 ms and 1.141 ms, even though the measured settle time is less than 0.8 ms. This extra buffer of about 14° ensures that the disk will not miss a rotation during sequential access when going from one track to the next.

Note that depending on the mapping of logical blocks (*LBNs*) to physical locations these blocks can appear to an application either sequential or non-contiguous. The choice is simply based on how a particular disk drive does its low-level logical-to-physical mapping. For example, a pair of sequential *LBNs* mapped to two different tracks or cylinders are still adjacent, according to our definition, as are two specific non-contiguous *LBNs* mapped to two nearby tracks.

Accessing successive adjacent disk blocks enables *semi-sequential* disk access [13, 27], which is the second-most efficient disk access method after pure sequential access. The delay between each access is equal to a disk head settle time, which is the minimum mechanical positioning delay the disk can offer. However, the semi-sequential access introduced previously utilizes only one of the adjacent blocks for efficiently accessing 2D data structures. This work shows how to use up to d adjacent blocks to improve access efficiency for multidimensional datasets.

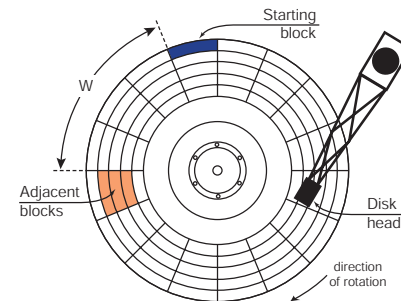


Figure 5: **Location of adjacent blocks.** $W = 67.5^\circ$ and $C = 3$.

4 Determining d

The previous section defined the adjacency relationship and identified the disk characteristics which enable access to adjacent blocks. We now describe two methods for determining the value of d . The first method we use analyzes the extracted seek curves and drive parameters to arrive at an estimate for C (and, by extension, d), and the second method empirically measures d directly. We evaluate and cross-validate both methods for a set of disk drives from two different vendors.

4.1 Experimental setup

All experiments described here are conducted on a two-way 1.7 GHz Pentium 4 Xeon workstation running Linux kernel 2.4.24. The machine has 1024 MB of

main memory and is equipped with one Adaptec Ultra160 Wide SCSI adapter connecting the disks. For our experiments, some of our disks have fewer platters than the maximum supported, which are listed in Table 1. Most enterprise drives are sold in families supporting a range of capacities, in which the only difference is the number of platters in the drive. Specifically, our Cheetah 36ES and Atlas 10k III have two platters, while the Atlas 10k IV, Atlas 10k V, Cheetah 10k.7, and Cheetah 15k.4 disks have only one platter. All but one disk have the same total capacity of 36.7 GB; the Cheetah 10k.7 is a 73 GB disk. Requests are issued to the disks via the Linux SCSI Generic driver and are timed using the CPU's cycle counter. All disks had their default cache mode page settings with both read and write cache enabled.

4.2 Seek measurements

To determine the proper values of C and d based on the disk's characteristics, we need to measure its seek profile. Since we do not have access to the disk firmware, we have to determine it empirically. We first obtain mappings of each *LBN* to its physical location, given by the $\langle \text{cylinder, head, sector} \rangle$ tuple. We use the SCSI address translation mode page (0x40h) and MODE SELECT command. With a complete layout map, we can choose a pair of tracks for the desired seek distance and measure how long it takes to seek from one to the other.

To measure seek time, we choose a pair of cylinders separated by the desired distance, issue a pair of read commands to sectors in those cylinders and measure the time between their completions. We choose a fixed *LBN* on the source track and successively change the value of the *LBN* on the second track, each time issuing a pair of requests, until we find at the lowest time between request completions. This technique is called the Minimal Time Between Request Completions (MTBRC) [37]. The seek time we report is the average of 6 trials of MTBRC, each with randomly-chosen starting locations spread across the entire disk. Note that the MTBRC measurement subtracts the time to read the second sector as well as bus and system overheads [37].

4.3 Seek profile analysis

The first method we use to determine C (and thus d) is based on analyzing the seek profile of a disk. Figure 6 shows seek profiles of the disk drives we evaluated for small cylinder distances. Note that extracted profiles have very similar shape, especially among drives from the same vendor. The one-cylinder seek time is the lowest, as expected. For distances of two and more cylinders, the seek time rises rapidly for a few cylinders, but then levels off for several cylinders before it experiences a large increase between distance of i and $i + 1$ cylinders.

After this inflection point, which is C , seek time rises gradually with increasing seek distance.

Note that the seek profile of some disks have more than one "plateau" and, thus, several possible values of C . When determining our value of C we chose the discontinuity point where seek times after the distance of C are at least 80% more than the one-cylinder seek time, while seek times of up to C are at most 60% larger than the one-cylinder seek time. Note that this is just one way of choosing the appropriate value of C . In practice, disk designers are likely to choose a value manually based on the physical disk parameters, just as they choose the value of track and cylinder skew. In either case, the choice of C is a trade-off between larger value of d (which increases the number of potential dimensions that can be accessed efficiently), and the efficiency of accesses to individual adjacent blocks.

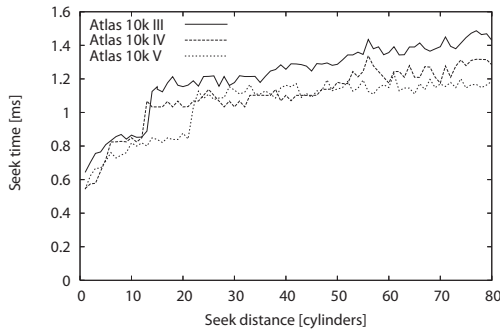
Using their measured seek curves, we now determine suitable values of C for six recent disk drives. Recall that C is the maximal seek distance in cylinders for which positioning time is (nearly) constant. Table 2 lists the value for each disk drive determined as the inflection point/discontinuity in the seek profile. The other pair of numbers in the table shows the percentage difference between seek time for distance of 1 cylinder and the distance of respectively C and $C+1$ cylinders, highlighting this discontinuity.

First, as expected, for more recent disk drives the value of C increases. Second, the difference between one-cylinder and C -cylinder seek times is about 50%. And finally, the difference in seek time between a one- and $C+1$ -cylinder seek is significant: between $1.7\times$ and $2\times$ the value of the one-cylinder seek.

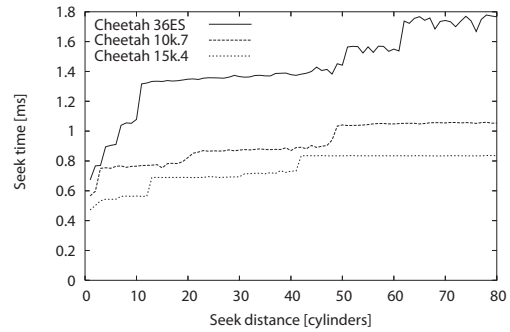
Once we have determined a value for C , we simply multiply by the number of surfaces in the disk to arrive at d . Figure 7 depicts the values of d for our disks. For each of the disks, we use the value of C from Table 2 and multiply it by the maximal number of surfaces each disk can have. From Table 1, for all but the Cheetah 36ES, $R = 8$. We plot the value of d as a function of year when the particular disk model was introduced. For years with multiple disks, we average the value of d across all analyzed disk models. Confirming our trend analysis, the value of d increases from 40 in year 2001 to almost 300 in year 2004. Recall that the value of d is proportional to the number of surfaces in the disk, R , and that the lower-capacity versions of the disk drives (such as those in our experiments) will have smaller values of d than those with more platters.

4.4 Measuring C directly

We now verify our previous method of determining C by measuring it directly. We measure the value for d rather than C , but, of course, C can easily be determined



(a) Recent Maxtor enterprise-class disks



(b) Recent Seagate enterprise-class disks

Figure 6: Measured seek profiles. Only the first 80 cylinders are shown.

Disk Model	Year	C	1-cyl seek diff. vs.	
			C-cyl	(C+1)-cyl
Atlas 10k III	2002	13	38%	76%
Atlas 10k IV	2003	12	52%	98%
Atlas 10k V	2004	21	55%	91%
Cheetah 36 ES	2001	10	59%	94%
Cheetah 10k.7	2004	49	44%	80%
Cheetah 15k.4	2004	42	55%	78%

Table 2: Estimated values of C based on seek profiles. The seek times compare the extracted value of 1-cylinder seek and C -cylinder seek. The last column lists the percentage increase between 1-cylinder and C -cylinder seek time.

by dividing d by the number of surfaces in the disk, R . We use the low-level layout model and the value of W to identify those blocks that are adjacent to the starting block. The experiment chooses a random starting block and a destination block which is i tracks away and is skewed by W degrees relative to the starting block.

We issue the two requests to the disk simultaneously and measure the response time of each one individually. If the difference in response time is equal to the settle time of the disk, then the two blocks are truly adjacent, and $i < d$. We increase i until the response time of the second request increases significantly, beyond the rotational period of the disk. This value of i is the maximum distance that the disk head can move and still access adjacent blocks without missing rotations, so $d = i - 1$.

Adding conservatism to the rotational offset, W , provides a useful buffer because of nondeterminism in the seek time. We found this to be especially true when experimenting with real disks, so our baseline values for W include an aggressive value of 10° for conservatism. Recall that the Atlas 10k III layout, for example, uses a buffer of 14° for track and cylinder skews.

Larger conservatism can increase the value of d at the expense of additional rotational latency and, hence, lower semi-sequential efficiency. Conceptually, this is

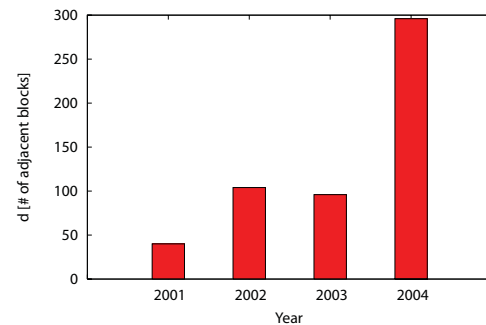


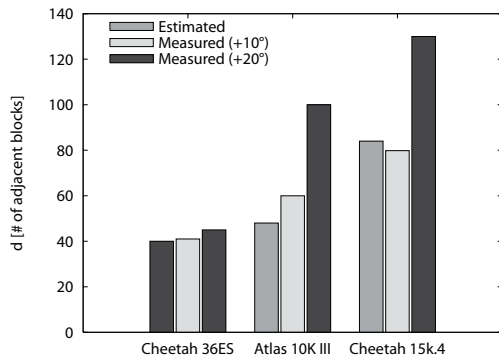
Figure 7: The trend in estimated value of d . The reported values are based on our estimated values of C multiplied by the disk's maximal number of surfaces. For years where we measured data from more than one disk model, we report the average value of d .

analogous to moving to the right along the seek profile past the discontinuity point. Larger values of d , in turn, allow mappings of data sets with many more dimensions, while maintaining the same efficiency for accesses to all $N - 1$ dimensions. Even though, more conservatism (and larger d) lowers the achieved semi-sequential bandwidth, it considerably increases the value of d as illustrated in Figure 8.

Figure 8(a) shows the comparison between the value of d based on our seek profile estimates of C reported in Table 2 and our measured values using the above approach. For each disk, we show three bars. The first bar, labeled “Estimated”, is the value of the estimated d . The second and third bar, labeled “Measured (+10°)” and “Measured (+20°)”, show the measured value with a conservatism of 10° and 20° , respectively, added to W . The “Estimated” values are based on our measurements from our disks, which have fewer platters than the models reported in Table 1. In contrast, the values reported in Figure 7 are based on disks with maximal capacities.

4.5 Eliminating rotational latency

A key feature of adjacent blocks is that, by definition, they can be accessed immediately after the disk head



(a) Comparison of estimated and measured d .

Disk Model	W	Extra	d	Access time
Atlas 10k III	68°	0°	25	0.95 ms
		10°	60	1.00 ms
		20°	100	1.25 ms
Cheetah 36ES	59°	0°	20	0.85 ms
		10°	35	1.00 ms
		20°	45	1.20 ms
Cheetah 15k.4	70°	0°	18	0.85 ms
		10°	80	0.85 ms
		20°	130	0.85 ms

(b) Increasing conservatism increases d and access time.

Figure 8: Comparison of estimated and measured d .

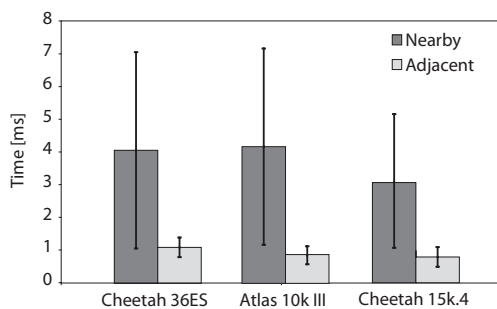


Figure 9: **Quantifying access times.** This graph compares the access times to blocks located within C cylinders. For the first two disks, the average rotational latency is 3 ms, for Cheetah 15k.4, it is 2 ms.

settles, without any rotational latency. To quantify the benefits of eliminating rotational latency, we compare adjacent access to simple nearby access within d tracks. Without explicit knowledge of adjacency, accessing each pair of nearby blocks will incur, on average, rotational latency of half a revolution, in addition to the seek time equivalent to the settle time. If these blocks are specifically chosen to be adjacent, then the rotational latency is eliminated and the access is much more efficient.

As shown in Figure 9, adjacent access outperforms nearby access by a factor of 4 thanks to the elimination of all rotational latency. Additionally, the access time for the nearby case varies considerably due to variable rotational latency, while the access time variability is much smaller for the Adjacent case; it is entirely due to the difference in seek time within the C cylinders, as depicted by the error bars.

5 Expressing adjacency to applications

The previous sections detailed the principles behind efficient access to d adjacent blocks and demonstrated that existing functions inside disk firmware (e.g., request schedulers) can readily identify and access these blocks. However, today's interfaces do not expose these blocks outside the disk. This section presents the method we

use for exposing adjacent blocks so that applications can use them for efficient access to multidimensional data. It first describes how individual disks can cleanly expose adjacent blocks, and then shows how to combine such information from individual disks comprising a logical volume and expose it using the same abstractions.

5.1 Exposing adjacent blocks

To allow for efficient access, the linear abstraction of disk drives sets an explicit contract between contiguous *LBNs*. To extend efficient access to adjacent blocks, we need to expose explicit relationships among the set of adjacent *LBNs* that are *non-contiguous*.

To expose the adjacency relationships, we need to augment the existing interface with one function, here called *GETADJACENT*. Given an *LBN*, this function returns a list of adjacent *LBNs* and can be implemented similarly to a *LBN*-to-physical address translation, i.e., a vendor-specific SCSI mode page accessed with the *MODE SELECT* command. The application need not know the reasons how or why the returned d disk blocks are adjacent, it just needs to have them identified through the *GETADJACENT* function.

A useful (conceptual) way to express the adjacency relationships between disk blocks is by constructing adjacency graphs, such as that shown in Figure 10. The graph nodes represent disk blocks and the edges connect blocks that are adjacent. The graph in the figure shows two levels of adjacency: the root node is the starting block, the nodes in the intermediate level are adjacent to that block, and the nodes in the bottom level are adjacent to the blocks in the intermediate level. Note that adjacent sets of adjacent blocks (i.e., those at the bottom level of the graph) overlap. For brevity, the graph shows only the first 6 adjacent blocks (i.e., $d = 6$), even though d is considerably larger for this disk, as described in Section 4. With the concept of adjacent blocks, applications can lay out and access multidimensional data with the existing 1D abstraction of the disk. This is pos-

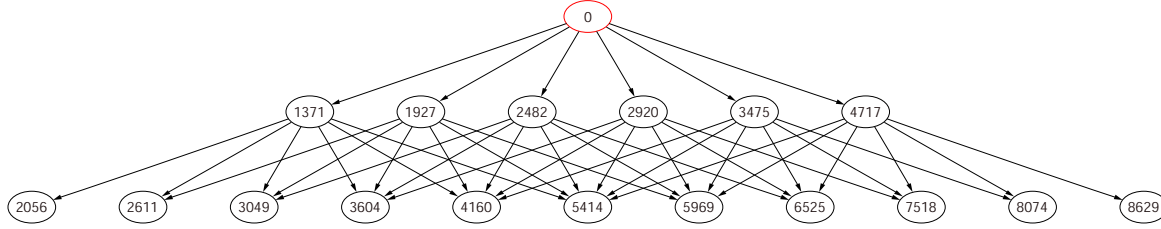


Figure 10: Adjacency graph for *LBN 0* of *Atlas 10k III*. Only two levels adjacent blocks are shown. The *LBNs* are shown inside nodes.

sible through explicit mapping of particular points in the data’s multidimensional space to particular *LBNs* identified by the disk drive.

5.2 Identifying adjacent blocks

Since current disk drives do not expose adjacent blocks and we do not have access to disk firmware to make the necessary modifications, we now describe an algorithm for identifying them in the absence of the proper storage interface functions. The algorithm uses a detailed model of low-level disk layout borrowed from a storage system simulator called DiskSim [8]. The parameters can be extracted from SCSI disks by previously published methods [25, 35, 37]. The algorithm uses two functions that abstract the disk-specific details of the disk model: *GETSKEW*(*lbn*), which returns the physical angle between the physical location of an *LBN* on the disk and a “zero” position, and *GETTRACKBOUNDARIES*(*lbn*), which returns the first and the last *LBN* at the ends of the track containing *lbn*.

For convenience, the algorithm also defines two parameters. First, the parameter *T* is the number of disk blocks per track and can be found by calling *GETTRACKBOUNDARIES*, and subtracting the low *LBN* from the high *LBN*. Of course, the value of *T* varies across zones of an individual disk, and will have to be determined for each call to *GETADJACENT*. Second, the parameter *W* defines the angle between a starting block and its adjacent blocks. This angle can be found by calling the *GETSKEW* function twice for two consecutive *LBNs* mapped to two different tracks and computing the difference; disks skew the mapping of *LBNs* on consecutive tracks by *W* degrees to account for settle time and to optimize sequential access.

The *GETADJACENT* algorithm, shown in Figure 11, takes as input a starting *LBN* (*lbn*) and finds the adjacent *LBN* that is *W* degrees ahead and *step* tracks away. Every disk block has an adjacent block within the *d* closest tracks, so the entire set of adjacent blocks is found by calling *GETADJACENT* for increasing values of *step* from 1 to *d*.

```

/* Find an LBN adjacent to lbn and step tracks away */
L := GETADJACENT(lbn, step) :
/* Find the required skew of target LBN */
target_skew := (GETSKEW(lbn) + W) mod 360

/* Find the first LBN in target track */
base_lbn := lbn + (step * T)
{low, high} := GETTRACKBOUNDARIES(base_lbn)

/* Find the minimum skew of target track */
low_skew := GETSKEW(low)

/* Find the offset of target LBN from the start of target track */
if (target_skew > low_skew) then
    offset_skew := target_skew - low_skew
else
    offset_skew := target_skew - low_skew + 360
end if

/* Convert the offset skew into LBNs */
offset_lbn := (offset_skew / 360) * T
RETURN(low + offset_lbn)

/* Find the physical skew of lbn, measured in degrees */
A := GETSKEW(lbn)

/* Find the boundaries of the track containing lbn */
{L, H} := GETTRACKBOUNDARIES(lbn)

T: Track length - varies across zones of the disk
W: Skew to add between adjacent blocks, measured in degrees

```

Figure 11: Algorithm for the *GETADJACENT* function.

5.3 Logical volumes

So far, we have discussed how to expose adjacent blocks to applications from a single disk drive. However, large storage systems combine multiple disks into logical volumes. From our perspective, a logical volume manager (LVM) adds nothing more than a level of indirection through mapping of a volume *LBN* (*VLBN*) to the *LBNs* of individual disks (*DLBN*). Given a set of adjacent blocks, an LVM can choose an explicit grouping of *LBNs* across all underlying *k* disks. The *d* *VLBNs* exposed via *GETADJACENT* are the adjacent blocks of a particular disk’s *DLBN* mapped to a given *VLBN* by the LVM. To an application, a multi-disk logical volume will appear as a (bigger and faster) disk, whose adjacent blocks set has cardinality *d*.

Since existing disks do not implement the `GETADJACENT` and `GETTRACKBOUNDARIES` functions, in our prototype implementation, a shim layer below our LVM extracts the information from the disk drives. It does so when the logical volume is initially created and provides these functions for the given disk. The LVM then stripes contiguous *VLBNs* across k individual disks and exposes to applications a set of d adjacent blocks in the *VLBN* space through the `GETADJACENT` function.

Much like other disk array logical volumes [6, 27], our LVM matches stripe units to track sizes for efficient sequential access. Our LVM exposes to applications the stripe unit size, T , through the `GETTRACKBOUNDARIES` function. It can adopt common RAID 1 and RAID 5 protection schemes and utilize multi-zone disks with defective blocks in a fashion similar to previous work [27].

For multi-zone disks, our LVM can either create multiple logical volumes, one for each zone, or create one logical volume that spans multiple zones. In the latter case, our LVM uses varies the value of T according to the number of sectors per track in the disk's zone to which the *VLBNs* is mapped. Put differently, in this approach, which we adopt for our experiments, a single logical volume has variable "stripe unit" size and our mappings of multidimensional data use the information exposed through the `GETTRACKBOUNDARIES` function to determine the proper mapping along the one dimension (see Section 6 for more details). Finally, d does not depend on the number of zones; it is strictly a function of track density (TPI) and the seek profile, which is fixed for a given disk and does not change with the location of the track.

6 Multidimensional data placement

This section demonstrates on 3D and 4D datasets how applications can utilize the adjacent blocks and the parameter T datasets onto disks in a way that preserves spatial locality. Through experiments with real disks and various workloads we show that this new mapping scheme outperforms existing schemes.

6.1 Data placement that preserves locality

To demonstrate the efficiency of accesses to adjacent blocks, we compare two existing mapping schemes for multidimensional data, *Naive* and *Hilbert*, with a new mapping scheme, called *MultiMap*. The *Naive* scheme linearizes the dataset along a chosen primary dimension (e.g., X or time). The *Hilbert* scheme orders the points according to their Hilbert curve values.

The *MultiMap* mapping scheme uses adjacent blocks to preserve spatial locality of multidimensional data on the disk(s). It first partitions the multidimensional data space into smaller chunks, called *basic cubes*, and then

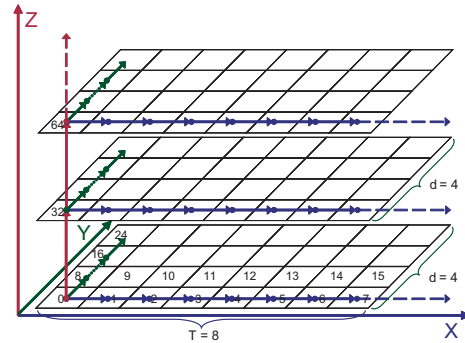


Figure 12: **Example of mapping a 3D dataset using *MultiMap*.** Each row in the graph represents a disk track with a length of 8 and each cell corresponds to a disk block whose lbn is the number inside the box. Suppose the value of d is 4. Dimension X is mapped to the sequential blocks on the same track. Dimension Y is mapped to the sequence of the first adjacent blocks. For example, lbn 8 is the first adjacent block of lbn 0 and lbn 16 is the first adjacent block of lbn 8, etc. Dimension Z is mapped to the sequence of the 4th adjacent blocks. For instance, lbn 32 is the 4th adjacent block of lbn 0 and lbn 64 is the 4th adjacent of lbn 32. In this way, *MultiMap* utilizes the adjacent blocks with different steps to preserve locality on the disk.

maps all the points within a basic cube into disk blocks on a single disk. Taking a 3D dataset as an example, *MultiMap* first maps points in the X dimension to T sequential *LBNs* so that accesses along X can take advantage of the full sequential bandwidth of the disk. Points along the Y dimension are mapped to the sequence of the first adjacent blocks. Lastly, the Z dimension is mapped to the sequence of d -th adjacent blocks, as Figure 12 shows. The sequence of adjacent blocks can be easily obtained by calling `GETADJACENT` repeatedly. With an LVM comprised of several disks, two "neighboring" basic cubes are mapped to two different disks and the basic cube becomes a multidimensional stripe unit.

MultiMap preserves the spatial locality in the sense that neighboring points in the geometric space will be stored in disk blocks that are adjacent to each other, allowing for access with minimal positioning cost. Since accessing the first adjacent block and the d -th adjacent block has the same cost, we can access up to d separate points that are equidistant from a starting point. This mapping preserves the spatial relationship that the next point along Y and the next point along Z are equidistant (in terms of positioning cost) to the same starting point. Retrieval along the X dimension result in efficient sequential access; retrieval along Y or Z result in semi-sequential accesses which are much more efficient than random or even nearby access, as shown in Figure 9.

Note that *MultiMap* is not a simple mapping of the 3D data set to the $\langle \text{cylinder}, \text{head}, \text{sector} \rangle$ tuple representing the coordinates of the physical blocks on disk. *MultiMap* provides a general approach for mapping N -dimensional data sets to adjacent blocks. For a given disk, the maximum number of dimensions that can be

mapped efficiently is limited by d for that disk, such that $N \leq \log_2(d) + 2$. As the focus of this work is the analysis of the principles behind multidimensional access and not the general data layout algorithm, we provide the generalized algorithm, its derivation, and the analysis of its limits elsewhere [31].

6.2 Experimental setup

Our experimental setup uses the same hardware configuration as in Section 4. Our prototype system consists of two software components running on the same host machine: a logical volume manager (LVM) and a database storage manager. In a production system, the LVM would likely reside inside a storage array system separate from the host machine. The prototype LVM implements the GETADJACENT algorithm and exports a single logical volume mapped across multiple disks. The database storage manager maps multidimensional datasets by utilizing the GETADJACENT and GETTRACKBOUNDARIES functions provided by the LVM. Based on the query type, it issues appropriate I/Os to the LVM, which then breaks these I/Os into proper requests to individual disks. Even when such requests are issued out of order, the disk firmware scheduler will reorder them to minimize the total positioning cost.

The datasets used for our experiments are stored on multiple disks in our LVM. Akin to commercial disk arrays, the LVM uses disks of the same type and utilizes only a part (slice) of the disk's total space [9]. The slices in our experiments are slightly less than half of the total disk capacity and span one or more zones.

Even though our LVM generates requests to all the disks during our experiments, we report performance results for only a single disk. The reason is that we examine average I/O response times, which depend only on the characteristics of a single disk drive. Using multiple drives improves the overall throughput, but does not affect the relative performance comparisons of the three mappings that our database storage manager implements: *Naive*, *Hilbert*, and *MultiMap*.

We evaluate two types of spatial queries. *Beam queries* are one-dimensional queries retrieving data points along lines parallel to the cardinal dimensions of the dataset. Range queries, called *p%-length cube queries*, fetch a cube with an edge length equal to the $p\%$ of the dataset's edge length.

6.3 Results using a 3D dataset

The 3D dataset used in this experiment contains $1024 \times 1024 \times 1024$ cells, where each cell maps to a distinct LBN of the logical volume and contains as many data points as can fit. We partition the space into chunks that each fit on a portion of a single disk. For both disks, *MultiMap* uses $d = 128$ and conservatism of 30° .

Beam queries. The results for beam queries are presented in Figure 13(a). We run beam queries along all three dimensions, X , Y , and Z , and the graphs show the average I/O time per cell (disk block). As expected, the *MultiMap* model delivers the best performance for all dimensions. It matches the streaming performance of *Naive* along X . More importantly, *MultiMap* outperforms *Hilbert* for Y and Z by 25%–35% and *Naive* by 62%–214% for the two disks. Finally, *MultiMap* achieves almost identical performance on both disks unlike *Hilbert* and *Naive*. That is because these disks have comparable settle times, which affect the performance of accessing adjacent blocks for Y and Z .

Range queries. The first set of three bars, labeled 1% in Figure 13(b), shows the performance of 1%-length cube queries expressed as their total runtime. As before, the performance of each scheme follows the trends observed for the beam queries. *MultiMap* improves the query performance (averaged across the two disks and the three query types) by 37% and 11% respectively compared to *Naive* and *Hilbert*. Both *MultiMap* and *Hilbert* outperform *Naive* as it cannot employ sequential access for range queries. *MultiMap* outperforms *Hilbert*, as *Hilbert* must fetch some cells from physically distant disk blocks, although they are close in the original dataset. These jumps make *Hilbert* less efficient compared to *MultiMap*'s semi-sequential accesses.

To examine the sensitivity of the cube query size, we also run 2%-length and 3%-length cube queries, whose results are presented in the second and third sets of bars in Figure 13(b). The trends are similar, with *MultiMap* outperforming *Hilbert* and *Naive*. The total run time increases because each query fetches more data.

6.4 Results using a 4D dataset

In earthquake simulation, we use a 3D grid to model the 3D region of the earth. The simulation computes the motion of the ground at each node in the grid, for a number of discrete time steps. The 4D simulation output contains a set of 3D grids, one for each step.

Our dataset is a $2000 \times 64 \times 64 \times 64$ grid modeling a 14 km deep slice of earth of a 38×38 km area in the vicinity of Los Angeles with a total size of 250 GB of data. 2000 is the total number of time steps. We choose time as the primary dimension for the *Naive* and the *MultiMap* schemes and partition the space into chunks that fit in a single disk.

The results, presented in Figure 14, exhibit the same trends as the 3D experiments. The *MultiMap* model again achieves the best performance for all beam and range queries. In Figure 14(a), the unusually good performance of *Naive* on Y is due to a particularly fortunate mapping that results in strided accesses that do not incur any rotational latency. The ratio of strides to track

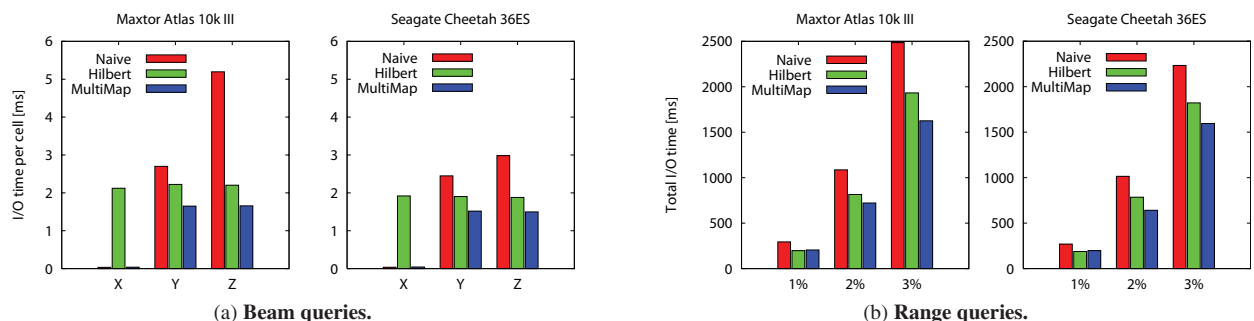


Figure 13: Performance using the 3D spatial dataset.

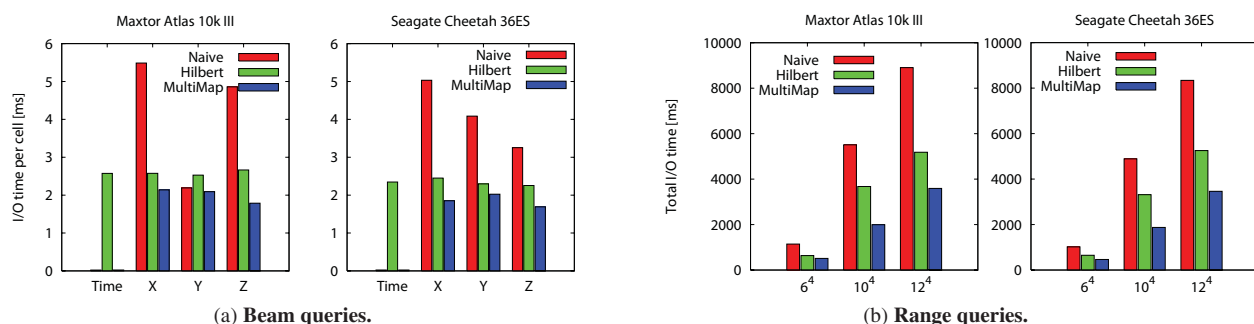


Figure 14: Performance using the 4D earthquake dataset.

sizes also explains the counterintuitive trend of the *Naive* scheme’s performance on the Cheetah disk where *Z* outperforms *Y*, and *Y* outperforms *X*. The range queries, shown in Figure 14(b), perform on both disks as expected from the 3D case. In summary, *MultiMap* is efficient for processing queries against spatio-temporal datasets, such as this earthquake simulation output, and is the only scheme that can combine streaming performance for time-varying accesses with efficient spatial access, thanks to the preservation of locality on disk.

7 Conclusion

The work presented here exploits disk drive technology trends. It improves access to multidimensional datasets by allowing the spatial locality of the data to be preserved in the disk itself. Through analysis of the characteristics of several state-of-the-art disk drives, we show how to efficiently access non-contiguous adjacent *LBNs*, which are hundreds of tracks away. Such accesses can be readily realized with the existing disk firmware functions and mappings of *LBNs* to physical locations.

Using our prototype implementation built with real, off-the-shelf disk drives, we demonstrate that applications can utilize streaming bandwidth for accesses along one dimension and efficient semi-sequential accesses in the other $N - 1$ dimensions. To the best of our knowledge, this is the first approach that can preserve spatial locality of stored multidimensional data, thus improving performance over current data placement techniques.

Acknowledgments

We thank John Bucy for his years of effort developing and maintaining the disk models that we use for this work. We thank the members and companies of the PDL Consortium (including APC, EMC, EqualLogic, Hewlett-Packard, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, and Sun) for their interest, insights, feedback, and support. We also thank IBM, Intel, and Seagate for hardware donations that enabled this work. This work is funded in part by NSF grants IIS-0133686 and CCR-0205544, by a Microsoft Research Grant, and by an IBM faculty partnership award.

References

- [1] K. A. S. Abdel-Ghaffar and A. E. Abbadi. Optimal Allocation of Two-Dimensional Data. *International Conference on Database Theory* (Delphi, Greece), pages 409-418, January 8-10, 1997.
- [2] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O’Hallaron, T. Tu, and J. Urbanic. High Resolution Forward And Inverse Earthquake Modeling on Terascale Computers. *ACM/IEEE Conference on Supercomputing*, page 52, 2003.
- [3] D. Anderson, J. Dykes, and E. Riedel. More than an interface: SCSI vs. ATA. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2003), pages 245–257. USENIX Association, 2003.
- [4] M. J. Atallah and S. Prabhakar. (Almost) Optimal Parallel Block Access for Range Queries. *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Dallas, Texas, USA), pages 205-215. ACM, 2000.

- [5] R. Bhatia, R. K. Sinha, and C.-M. Chen. Declustering Using Golden Ratio Sequences. *ICDE*, pages 271–280, 2000.
- [6] P. M. Chen and D. A. Patterson. *Maximizing performance in a striped disk array*. UCB/CSD 90/559. Computer Science Div., Department of Electrical Engineering and Computer Science, University of California at Berkeley, February 1990.
- [7] P. M. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. *ACM SIGMOD International Conference on Management of Data*, pages 259–270. ACM Press, 1998.
- [8] The DiskSim Simulation Environment (Version 3.0). <http://www.pdl.cmu.edu/DiskSim/index.html>.
- [9] EMC Corporation. *EMC Symmetrix DX3000 Product Guide*, http://www.emc.com/products/systems/DMX_series.jsp, 2003.
- [10] C. Faloutsos. Multiattribute hashing using Gray codes. *ACM SIGMOD*, pages 227–238, 1986.
- [11] C. Faloutsos and P. Bhagwat. Declustering Using Fractals. *International Conference on Parallel and Distributed Information Systems* (San Diego, CA, USA), 1993.
- [12] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, **30**(2):170–231, 1998.
- [13] G. G. Gorbatenko and D. J. Lilja. *Performance of two-dimensional data models for I/O limited non-numeric applications*. Laboratory for Advanced Research in Computing Technology and Compilers Technical report ARCTiC-02-04. University of Minnesota, February 2002.
- [14] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. *ICDE*, pages 152–159. IEEE Computer Society, 1996.
- [15] J. Gray, D. Slutz, A. Szalay, A. Thakar, J. vandenBerg, P. Kunszt, and C. Stoughton. *Data Mining the SDSS Skyserver Database*. Technical report. Microsoft Research, 2002.
- [16] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *ACM SIGMOD*, pages 47–57, 1984.
- [17] D. Hilbert. Über die stetige Abbildung einer Linie auf Flächenstück. *Math. Ann.*, **38**:459–460, 1891.
- [18] H. V. Jagadish, L. V. S. Lakshmanan, and D. Srivastava. Snakes and sandwiches: optimal clustering strategies for a data warehouse. *SIGMOD Rec.*, **28**(2):37–48. ACM Press, 1999.
- [19] N. Koudas, C. Faloutsos, and I. Kamel. Declustering Spatial Databases on a Multi-Computer Architecture. *International Conference on Extending Database Technology* (Avignon, France), 1996.
- [20] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. *Analysis of the clustering properties of Hilbert space-filling curve*. Technical report. University of Maryland at College Park, 1996.
- [21] Office of Science Data-Management Workshops. *The Office of Science Data-Management Challenge*. Technical report. Department of Energy, 2005.
- [22] J. A. Orenstein. Spatial query processing in an object-oriented database system. *ACM SIGMOD*, pages 326–336. ACM Press, 1986.
- [23] S. Prabhakar, K. Abdel-Ghaffar, D. Agrawal, and A. E. Abbadi. Efficient Retrieval of Multidimensional Datasets through Parallel I/O. *International Conference on High Performance Computing*, page 375. IEEE Computer Society, 1998.
- [24] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.
- [25] J. Schindler and G. R. Ganger. *Automated disk drive characterization*. Technical report CMU-CS-99-176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.
- [26] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.
- [27] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: a disk array volume manager for orchestrated use of disks. *Conference on File and Storage Technologies* (San Francisco, CA, 31 March–02 April 2004). USENIX Association, 2004.
- [28] S. W. Schlosser, J. Schindler, A. Ailamaki, and G. R. Ganger. *Exposing and exploiting internal parallelism in MEMS-based storage*. Technical Report CMU-CS-03-125. Carnegie-Mellon University, Pittsburgh, PA, March 2003.
- [29] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. *Winter USENIX Technical Conference* (Washington, DC, 22–26 January 1990), pages 313–323, 1990.
- [30] M. Shao, J. Schindler, S. W. Schlosser, A. Ailamaki, and G. R. Ganger. Clotho: Decoupling memory page layout from storage organization. *International Conference on Very Large Databases* (Toronto, Canada, 31 August–2 September 2004), 2004.
- [31] M. Shao, S. W. Schlosser, S. Papadomanolakis, J. Schindler, A. Ailamaki, C. Faloutsos, and G. R. Ganger. *MultiMap: Preserving disk locality for multidimensional datasets*. Carnegie Mellon University Parallel Data Lab Technical Report CMU-PDL-05-102. March 2005.
- [32] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. *International Conference on Very Large Databases*, pages 522–531. Morgan Kaufmann Publishers Inc., 1996.
- [33] K. Stockinger, D. Düllmann, W. Hoschek, and E. Schikuta. Improving the Performance of High-Energy Physics Analysis through Bitmap Indices. *Database and Expert Systems Applications*, pages 835–845, 2000.
- [34] T. Stoehr, H. Maertens, and E. Rahm. Multi-Dimensional Database Allocation for Parallel Data Warehouses. *International Conference on Very Large Databases*, pages 273–284. Morgan Kaufmann Publishers Inc., 2000.
- [35] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. *Microbenchmark-based extraction of local and global disk characteristics*. Technical report CSD-99-1063. University of California at Berkeley, 13 June 2000.
- [36] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling algorithms for modern disk drives. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Nashville, TN, 16–20 May 1994), pages 241–251. ACM Press, 1994.
- [37] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146–156, May 1995.
- [38] H. Yu, D. Agrawal, and A. E. Abbadi. Tabular placement of relational data on MEMS-based storage devices. *International Conference on Very Large Databases* (Berlin, Germany, 09–12 September 2003), pages 680–693, 2003.
- [39] H. Yu, K.-L. Ma, and J. Welling. Automated Design of Multidimensional Clustering Tables for Relational Databases. *International Conference on Very Large Databases*, 2004.
- [40] H. Yu, K.-L. Ma, and J. Welling. A Parallel Visualization Pipeline for Terascale Earthquake Simulations. *ACM/IEEE conference on Supercomputing*, page 49, 2004.
- [41] Y. Zhao, P. M. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. *ACM SIGMOD International Conference on Management of Data* (Tucson, AZ, 13–15 May 1997). Published as *SIGMOD Record*, **26**(2):159–170. ACM, 1997.

Database-Aware Semantically-Smart Storage

Muthian Sivathanu*, Lakshmi N. Bairavasundaram†,

Andrea C. Arpaci-Dusseau†, and Remzi H. Arpaci-Dusseau†

* Google Inc. † Computer Sciences Department, University of Wisconsin, Madison

Abstract

Recent research has demonstrated the potential benefits of building storage arrays that understand the file systems above them. Such “semantically-smart” disk systems use knowledge of file system structures and operations to improve performance, availability, and even security in ways that are precluded in a traditional storage system architecture.

In this paper, we study the applicability of semantically smart disk technology underneath database management systems. For three case studies, we analyze the differences when building database-aware storage. We find that semantically-smart disk systems can be successfully applied underneath a database, but that new techniques, such as log snooping and explicit access statistics, are needed.

1 Introduction

Processing power is increasing in modern storage systems. For example, the Symmetrix storage array, a high-end RAID from EMC, contains nearly 100 processors and up to 256 GB of memory [9]. Unfortunately, the ability to leverage the computational power within traditional storage systems has been limited due to its narrow block-based interface [8, 10]. With protocols such as SCSI, storage arrays receive only the simplest of commands: read or write a given range of blocks. Hence, the storage system has no knowledge of how it is being used, *e.g.*, whether two blocks are a part of the same file, or even whether a given block is live or dead.

To bridge this information gap, recent research has proposed the idea of a *semantically smart disk system* [30] that either learns of or is embedded with knowledge of the file system using it. This semantic information within the storage system allows vendors to build more functional, reliable, higher-performing, and secure storage systems. For example, by exploiting knowledge of file and directory structures, a storage system can deliver improved data availability under failure [29].

Previous research on semantically smart disk systems [3, 28, 29, 30] has assumed that a commodity file

system (*e.g.*, Linux ext2, Linux ext3, NetBSD FFS, Windows FAT, or Windows NTFS) is interacting with the disk. In this paper, we explore techniques for semantically-smart disk systems to operate beneath database management systems (DBMS). Given that database systems form a significant and important group of clients of storage systems, we would like to see if the benefits of semantically smart storage can be applied to this realm.

Whether operating beneath a file system or a database, a semantically smart disk system performs similar operations, such as tracking which file or table a particular block has been allocated to. However, a DBMS tracks different information and organizes its data on disk differently than a file system does. For example, most file systems record within each file’s metadata certain statistics, such as the most recent access and modified time. Given that a DBMS is more specialized, it does not track these general statistics. Second, in file system workloads, the directory structure tends to be a reasonable approximation of semantic groupings; that is, users place related files together in a single directory. However, in a DBMS, the semantic grouping across different tables and their indexes is dynamic, depending upon the query workload.

Our general finding is that these differences are fundamental enough to require changes for semantically smart storage. To build database-aware storage, we investigate two techniques that were not required for file systems. First, we explore *log snooping*, in which the storage system observes the write-ahead log (WAL) records written by the DBMS; by monitoring this log, the storage system can observe every operation performed by the DBMS before its effect reaches the disk. Second, we explore the benefits of having the DBMS explicitly *gather access statistics* and write these statistics to storage. We find that it is relatively simple to add these statistics to a DBMS.

To investigate database-aware storage, we implement and analyze three case studies that have been found to work well underneath file systems. First, we study how to improve storage system availability with D-GRAID, a RAID system that degrades gracefully under failure [29]. Second, we implement a DBMS-specialized version of FADED, a storage system that guarantees that data is unrecoverable once the user has deleted it [28]. Finally, we explore how to improve second-level storage-array cache

*Work done while at the University of Wisconsin-Madison

hit rates with a technique known as X-RAY [3].

Our experience indicates that semantically-smart disks can work well underneath database systems. In some cases, database systems are a better fit than file systems for semantically-smart storage, such as for secure delete [28]. In this case, the presence of the transactional semantics in a DBMS allows the disk to more accurately track dynamic information. As a result, functionality that requires absolutely correct inferences can be implemented without changing the DBMS; in contrast, this same functionality required changes to the file system. However, for two of the case studies, D-GRAID and X-RAY, we find that a DBMS does not supply all of the desired access information to the storage system. As a result, better results are obtained if we slightly modify the DBMS.

The rest of this paper is organized as follows. In Section 2, we review related work in database-aware storage and discuss the advantages and disadvantages of semantically-smart disks. In Section 3, we describe the general techniques needed for a semantic disk to extract information from the DBMS. In Sections 4, 5, and 6, we present our case studies. Finally, we discuss the range of useful techniques in Section 7 and conclude in Section 8.

2 Background

Placing more intelligence in disk systems to help database systems has come in and out of favor over the years. For a summary of work in this area, see Keeton's dissertation [17], page 162. One of the earliest examples is the idea of "logic per track" devices proposed in 1970 [31]; for example, given a disk with some computational ability per head, a natural application is to filter data before it passes through the rest of the system.

Later, the idea of database-specific machines was refuted, for example in 1983 by Boral and Dewitt [5]. The primary reason for the failure of such approaches was that they often required non-commodity components and were outperformed as technology moved ahead; worse, database vendors did not wish to rewrite their substantial code base to take advantage of specific features offered by certain specialized architectures.

However, as processing power has become faster and cheaper, the idea of "active disks" has come into focus once more. Recent work includes that by Acharya *et al.* [1] and Riedel *et al.* [25]; in both efforts, portions of applications are downloaded to disks, thus tailoring the disk to the currently running program. Much of this research focuses on exactly how to partition applications across host and disk CPUs to minimize data transferred.

In contrast to much of this previous work, the semantically-smart approach does not require specialized hardware components or sophisticated programming environments [3, 28, 29, 30]. High-end storage arrays are a good match for this technology, as they often have multi-

ple processors and vast quantities of memory. However, building semantic knowledge of higher-level systems into a storage array has both benefits and drawbacks.

The main benefit of the semantic-disk approach is that it increases functionality; placing high-level semantic knowledge within the storage system enables new systems that require both the low level control available within the storage array, and high level knowledge about the DBMS; such systems are precluded in traditional storage architectures. For example, previous research has shown that semantic disks can improve performance with better layout and caching [3, 30], can improve reliability [29], and can provide additional security guarantees [28].

However, the semantically-smart approach also leads to a few concerns. One concern is that too much processing will be required within the disk system. However, many researchers have noted that the trend is of increasing intelligence in disk systems [1, 25]. Indeed, modern storage arrays already exhibit the fruits of Moore's Law and the EMC Symmetrix storage server can be configured with up to 100 processors and 256 GB of RAM [9]. These resources are not idle, but nonetheless hint at the relative simplicity of adding more intelligence.

A second concern is that placing semantic knowledge within the disk system ties the disk system too intimately to the file system or DBMS above. For example, if the DBMS on-disk structure changes, the storage system may have to change as well. In file systems, on-disk formats rarely change; for example, the format of the ext2 file system has not significantly changed in its 10 years of existence [30], and current modifications take great pains to preserve full backwards compatibility with older versions of the file system [33]. In the case of a DBMS, format changes are more of a concern. To gain some insight on how often a storage vendor would have to deliver "firmware" updates in order to keep pace with DBMS-level changes, we studied the development of Postgres [24] looking for times in its revision history when a dump/restore was required to migrate to the new version. We found that a dump/restore was needed every 9 months on average, more frequent than we expected. However, in commercial databases that store terabytes of data, requiring a dump/restore to migrate is less tolerable to users; indeed, more recent versions of Oracle go to great lengths to avoid on-disk format changes.

A final concern is that the storage system must have semantic knowledge of each layer, whether a file system or a DBMS, that could possibly run upon it. Fortunately, there are only a few file systems and database systems that would need to be supported to cover a large fraction of the market. Further, much of the functionality in a semantic disk is independent of the layer above; thus, only a small portion of the code needs to handle issues that are specific to each file system or DBMS. Finally, if a storage

vendor wants to reduce the burden of supporting many different database platforms, they can target a single important database (e.g., Oracle) and just provide standard RAID functionality for other systems. Interestingly, high-end RAID systems already perform a bare minimum of semantically-smart behavior. For example, storage systems from EMC can recognize an Oracle data block and provide an extra checksum to assure that a block write (comprised of multiple sector writes) reaches disk atomically [6]. In summary, storage vendors are already willing to commit resources to support database technology.

3 Database-Aware Techniques

To implement powerful functionality, a storage system can leverage higher-level semantic information about the file system or DBMS that is running on top. In this section, we describe the types of information a semantic disk requires underneath a DBMS, and discuss how such information can be acquired. Database-specific semantic information can be broadly categorized into two types: static and dynamic.

Since our experience has primarily been with the Predator DBMS [27] built upon the SHORE storage manager [19], we illustrate our techniques with specific examples from Predator; however, we believe the techniques are general across other database systems.

3.1 Static information

Static information is comprised of facts about the database that do not change while the database is running. The storage system can obtain static information either by having such knowledge embedded in its firmware or by having it explicitly communicated through an out-of-band channel once during system installation.

In most cases, static information describes the *format of on-disk structures*. For example, by knowing the format of the database log record, the semantic disk can observe each update operation to disk; by knowing the structure of B-Tree pages, the disk can determine which are internal pages versus leaf pages; finally, by understanding the format of data pages, the semantic disk can perform operations such as scanning the page to find “holes” when byte ranges are deleted. In other cases, static information describes the *location of on-disk structures*. For example, in Predator, knowing the names and IDs of *system catalog tables* such as the *RootIndex* and the *_SINDXS* table is useful.

3.2 Dynamic information

Dynamic information pertains to information about the DBMS that continually changes during operation. Examples of dynamic information include the particular set of disk blocks allocated to a certain table or whether a given disk block belongs to a table or to an index. Unlike static information, dynamic information needs to be continually

tracked by the disk. To track dynamic information, a semantic disk utilizes static information about data structure formats to monitor changes to key data structures; these changes are then correlated to the higher level operations that could cause these changes.

Unfortunately, since both file systems and databases buffer and reorder writes, performing an accurate inference of higher level operations can be quite complex [28, 29]. To solve this problem, we use the technique of *log snooping*, in which the storage system observes the log records written out by the DBMS. With log snooping, the storage system leverages the fact that the database uses a write-ahead log (WAL) to track every operation that changes on-disk contents. Because of the WAL property, the log of an operation reaches disk *before* the effect of the operation; this strong ordering guarantee makes inferences underneath a DBMS accurate and straightforward.

Our implementation of log snooping is as follows. We assume that each log record contains a Log Sequence Number (LSN) [20]; the LSN is usually the byte offset of the start of that record in the log volume. The LSN allows the semantic disk to accurately infer the exact ordering of events that occurred in the database, even in the presence of group commits that can cause log blocks to arrive out of order. To order events, the disk maintains an *expected LSN* pointer, which is the LSN of the next log record expected to be seen by the disk; thus, when the semantic disk receives a write request to a log block, it knows exactly where in the block to look for the next log record. The semantic disk then processes that log record and advances the expected LSN pointer to point to the next record. Thus, even when log blocks arrive out of order, the semantic disk utilizes the LSN ordering to process the blocks in order; log blocks arriving out of order are *deferred* until the expected LSN reaches that block.

We now describe in more detail how our implementation of database-aware storage uses log snooping to infer four important pieces of dynamic information: transaction status, block ownership, block type, and relationships between blocks. We then describe the importance of a final piece of dynamic information: access statistics.

3.2.1 Transaction Status

A basic piece of dynamic information is the current state of each transaction that has been written to disk. Each transaction can be either *pending* or *committed* and a pending transaction may later be aborted. When performing work associated with a transaction, a semantic disk can choose to *pessimistically* recognize only committed transactions, or it can *optimistically* begin work on pending transactions as well. There are trade-offs to both the pessimistic and optimistic approaches.

The pessimistic approach is most appropriate when the semantic disk implements functionality that requires correctness. For example, when implementing secure delete

(Section 5), the semantic disk cannot shred data belonging to a pending transaction, given that the transaction may abort and the DBMS require the data again. However, the pessimistic approach will often have worse performance than the optimistic approach, since the pessimistic version must delay work and may require a significant amount of buffering. The optimistic approach is most beneficial when aborts are rare and the DBMS implements *group commits* (and may thus delay committing individual transactions for a long period).

Determining the status of each transaction is straightforward with log snooping. When the semantic disk observes that a new log record has been written, it adds it to a list of “pending” transactions; when the disk observes a `commit` record in the log, it determines which transactions have committed and moves them to a “committed” list.

3.2.2 Block Ownership

It is useful for a semantic disk to understand the logical grouping of blocks into tables and indices; this involves associating a block with the corresponding table or index *store* that logically *owns* the block. Performing this association in the semantic disk is relatively straight forward; since the effect of allocating a block must be recoverable, the DBMS first logs the operation before performing the allocation. Therefore, when the semantic disk later observes traffic to a disk block, it is simple to associate that block with the owning table or index. As we show later, in some cases it is sufficient for the semantic disk to map blocks to the store ID of the owning table, whereas in other cases, is useful for the semantic disk to further map the store ID to the actual table (or index) name.

For example, when allocating a block, SHORE writes out a `create_ext` log record with the block number and the ID of the owning store. When the semantic disk observes this log entry, it records the block number and store ID in an internal `block_to_store` hash table.

To further map the store ID to the actual table or index name, the disk uses static knowledge of the *system catalog tables*. In Predator, this mapping is maintained in a B-Tree called the *RootIndex*, whose logical store ID is statically known. Thus, when the disk observes `btree_add` records in the log with the *RootIndex* ID, the semantic disk is able to identify newly created mappings and add them to a `store_to_name` hash table.

3.2.3 Block Type

Another piece of useful information for a semantic disk is the *type* of a store (or a block); for example, whether a block is a data page or an index page. To track this information, the semantic disk again watches updates to the system catalog tables, the names of which are part of the static information known to the disk.

For example, in Predator, the `_SINDXS` table contains all indexes in the database; each tuple in `_SINDXS` contains the name of the index, the name of the table, and the attribute on which the index is built. The semantic disk detects inserts to this table by looking for the appropriate `page_insert` records in the log. The semantic disk is then able to determine whether a given block is part of a table or of an index by looking up its owning store in information derived from the `_SINDXS` table.

3.2.4 Block Relationships

A third type of useful information consists of the relationships across different blocks. One of the most useful relationships for a semantic disk to know is that between a table and the set of indices built on the table.

As stated above, in Predator, the association between indices and tables is kept the `_SINDXS` catalog table. Thus, a semantic disk can consult information derived from the `_SINDXS` table to associate a given table with its indices, or vice versa.

3.2.5 Access Patterns

In addition to the previous dynamic information, it is also useful for a semantic disk to know how tables and indexes are being accessed in the current workload. Although transaction status, block ownership, block type, and block relationships can be inferred relatively easily with log snooping, these access patterns are more difficult to infer.

Inferring access patterns was found to be relatively easy underneath a general-purpose file system [3, 30]. For example, the fact that a certain set of files lies within a directory implicitly conveys information to the storage system that those files are likely to be accessed together. Similarly, most file systems track the last time each file was accessed and periodically write this information to disk.

Although some modern database systems do track access statistics for performance diagnosis, the statistics are gathered at relatively coarse granularity; for example, the Automatic Workload Repository in Oracle 10g maintains access statistics [21].

Our experience has revealed that it would be useful for the DBMS to track three different types of statistics. Because this information is only used to optimize behavior, the DBMS can write the statistics periodically to disk (perhaps in additional catalog tables) without being transactional and thus can avoid the logging overhead.

The most basic statistic for the DBMS to communicate with the semantic disk is the *access time* of a particular block or table. This particular statistic is useful both in its own right and because it can be used to derive other statistics. A second useful statistic summarizes the *access correlation* between entities such as tables and indexes; for example, the DBMS could record for each query, the set of tables and indexes accessed. These correlation statistics

capture the semantic groupings between different tables and is useful for collocating related tables. Finally, a third useful statistic tracks *access counts*, such as the number of queries that accessed a given table over a certain duration. This piece of information conveys the importance of various tables and indexes.

3.3 Case Studies

The actual static or dynamic information required within a database-aware disk depends upon the functionality that the disk is implementing. Therefore, we investigate a number of case studies that were previously implemented underneath of file systems. First, we investigate D-GRAID, a RAID system that degrades gracefully under failure [29]. Second, we implement a FADED, which guarantees that data is unrecoverable once the user deletes it [28]. Finally, we explore X-RAY, which implements a second-level storage-array cache [3].

4 Partial Availability with D-GRAID

Our first case study is to implement D-GRAID [29] underneath a DBMS. D-GRAID is a semantically-smart storage system that lays out blocks in a way that ensures graceful degradation of availability under unexpected multiple failures. Thus, D-GRAID enables continued operation of the system instead of complete unavailability under multiple failures. Previous work has shown that this approach significantly improves the availability of file systems [29].

In this section, we begin by reviewing the motivation for partial availability and D-GRAID. Next, we summarize our past experience when implementing D-GRAID underneath file systems. We then describe our techniques for implementing D-GRAID underneath a DBMS. Finally, we evaluate our version of D-GRAID and discuss its lessons.

4.1 Motivation

The importance of data availability cannot be over emphasized, especially in settings where downtime can cost millions of dollars per hour [18, 23]. To cope with failures, file systems and database systems store data in RAID arrays [22], which employ redundancy to automatically recover from a small number of disk failures.

Existing RAID schemes do not effectively handle catastrophic failures, that is, when the number of failures exceeds the tolerance threshold of the array (usually 1). Multiple failures occur due to two primary reasons. First, faults are often correlated [12]; a single controller fault or other component error can render a number of disks unavailable [7]. Second, system administration is the main source of failure in systems [11]. A large percentage of human failures occur during maintenance, where “the maintenance person typed the wrong command or

unplugged the wrong module, thereby introducing a double failure” (page 6) [11].

Under such extra failures, existing RAID schemes lead to complete unavailability of data until the contents of the array are restored from backup. This effect is especially severe in large arrays; for example, even if 30 out of 32 disks (roughly 94%) in a RAID-5 array are fully operational, the disk system (and consequently, the database) is completely unavailable.

This “availability cliff” arises because traditional storage systems employ simplistic layout techniques such as striping, that are oblivious of the semantic importance of blocks or relationships across blocks; when excess failures occur, the odds of semantically-meaningful data (e.g., a table) remaining available are low. Furthermore, because modern storage arrays export abstract logical volumes which appear like a single disk [9, 34], the file system or DBMS has no control over data placement and cannot ensure that semantically-meaningful data remains after a single disk failure.

4.2 Filesystem-Aware D-GRAID

The basic goal of D-GRAID [29] is to make *semantically meaningful* fragments of data available under failures, so that workloads that access only those fragments can still run to completion, oblivious of data loss in other parts of the file system. By working on top of any redundancy technique (e.g., RAID-1), D-GRAID provides graceful degradation when the number of failures exceed the tolerance threshold of the particular redundancy technique. When we implemented D-GRAID under a file system, we found three layout techniques to be important.

First, *fault-isolated data placement* is needed to ensure that semantic fragments remain available in their entirety. Under fault isolated placement, an entire semantic fragment is colocated within a single disk. We found that, for file system workloads, a reasonable semantic fragment consists of either a single file in its entirety (i.e., its data blocks, its inode block, and potentially its indirect blocks) or all of the files in a single directory.

Second, *selective replication* is needed to ensure that essential meta-data and data that is always required is very likely to be available. In the file system context, this essential meta-data was found to consist of all directories (i.e., data and inode blocks) and the structures of the file system (i.e., superblock and bitmap blocks); the essential data was found to be the system binaries kept in known directories (e.g., in `/usr/bin`, `/bin`, and `/lib`).

Third, *access-driven diffusion* in which popular data is striped across disks, is needed to improve throughput when a large file is placed on a single disk. We found that popular data could be dynamically identified by tracking logical segments without semantic knowledge; thus, access-driven diffusion can be implemented in the same manner whether beneath a file system or a DBMS.

4.3 Database-Aware D-GRAID

We now describe our techniques for implementing D-GRAID underneath a DBMS. First, we explore two techniques for fault-isolated data placement that target widely different database usage patterns: moderately-sized tables which can use coarse-grained fragmentation and large tables which must use fine-grained fragmentation. Second, we explore the structures that need to be selectively replicated. Third, we describe our implementation of access-driven diffusion. Finally, we describe *infallible writes*, a new technique that was not required for file systems.

When identifying semantic fragments, there are two fundamental differences under a DBMS versus a file system. First, in a DBMS, one is more likely to find extremely large tables that will not fit within a single disk. Therefore, we describe our techniques separately for moderately-sized tables, which can use coarse-grained fragmentation and fit an entire table within a disk, and for very large tables, which must use fine-grained fragmentation and stripe tables and indexes across multiple disks. Second, in a DBMS, the queries being performed directly impact which tables and indexes are accessed together. Therefore, we describe how semantic groupings are affected by three popular types of queries: scans, index lookups, and joins.

4.3.1 Fault-Isolated Placement: Coarse-Grained

The simplest case occurs when the database contains a large number of moderately-sized tables; in this situation, a semantic fragment can be defined in terms of an entire table. We now present layout strategies for improved availability for each query type given this scenario.

A. Scans: Many queries, such as selection queries that filter on a non-indexed attribute or aggregate queries on a single table, involve a sequential scan of one entire table. Since a scan requires the entire table to be available in order to succeed, a simple choice of a semantic fragment is the set of all blocks belonging to a table; thus, an entire table is placed within a single disk, so that when failures occur, a subset of tables are still available in their entirety, and therefore scans just involving those tables will continue to operate oblivious of failure.

B. Index lookups: Index lookups form another common class of queries. When a selection condition is applied based on an indexed attribute, the DBMS looks up the corresponding index to find the appropriate tuple record IDs, and then reads the relevant data pages to retrieve the tuples. Since traversing the index requires access to multiple pages in the index, collocation of a whole index improves availability. However, if the index and table are viewed independently for placement, an index query fails if either the index or the table is unavailable, decreasing availability. Thus, a better strategy to improve availability is to collocate a table with its indexes. We call the latter strategy *dependent index placement*.

C. Joins: Many queries involve joins of multiple tables. Such queries typically require all the joined tables to be available, in order to succeed. To improve availability of join queries, D-GRAID collocates tables that are likely to be joined together into a single semantic fragment, which is then laid out on a single disk. Identification of such “join groups” requires extra access statistics to be tracked by the DBMS.

For our implementation, we modified the Predator DBMS to record the set of stores (tables and indexes) accessed for each query and to construct a matrix that indicates the access correlation between each pair of stores. This information is written to disk periodically (once every 5 seconds). These modifications to Predator are relatively straight-forward, involving less than 200 lines of code. D-GRAID then uses this information to collocate tables that are likely to be accessed together.

4.3.2 Fault-Isolated Placement: Fine-Grained

While collocation of entire tables and indexes within a single disk provides enhanced availability, a single table or index may be too large to fit within a single disk, even though disk capacities are roughly doubling every year [13]. In such a scenario, we require a fine-grained approach to semantic fragmentation. In this approach, D-GRAID stripes tables and indexes across multiple disks (similar to a traditional RAID array), but adopts new techniques to enable graceful degradation, as detailed below.

A. Scans: Scans fundamentally require the entire table to be available, and thus any striping strategy will impact availability of scan queries. To help availability, a hierarchical approach is possible: a large table can be split across the minimal number of disks that can hold it, and the disk group can be treated as a logical fault-boundary; D-GRAID can be applied over such logical fault-boundaries. Alternatively, if the database supports approximate queries [15], it can provide partial availability for scan queries even with missing data.

B. Index lookups: With large tables, index-based queries are likely to be more common. For example, an OLTP workload such as TPC-C normally involves index lookups on a small number of large tables. These queries do not require the entire index or table to be available. D-GRAID uses two simple techniques to improve availability for such queries. First, the internal pages of the B-tree index are aggressively replicated, so that a failure does not take away, for instance, the root of the B-tree. Second, an index page is collocated with the data pages corresponding to the tuples pointed to by the index page. For this collocation, D-GRAID uses a probabilistic strategy; when a leaf index page is written, D-GRAID examines the set of RIDs contained in the page, and for each RID, determines which disk the corresponding tuple is placed in. It then places the index page on the disk which has the greatest number of matching tuples. Note that we assume the table

is clustered on the index attribute; page-level collocation may not be effective in the case of non-clustered indexes.

C. Joins: Similar to indexes, page-level collocation can also be applied across tables of a join group. For such collocation to be feasible, all tables in the join group should be clustered on their join attribute. Alternatively, if some tables in the join group are “small”, they can be replicated across disks where the larger tables are striped.

4.3.3 Selective Replication

There are some data structures within a DBMS that must be available for *any* query in the system to be able to run. For example, system catalogs (that contain information about each table and index) are frequently consulted; if such structures are unavailable under partial failure, the fact that most data remains accessible is of no practical use. Therefore, D-GRAID aggressively replicates the system catalogs and the *extent map* in the database that tracks allocation of blocks to stores. In our experiments, we employ 8-way replication of important meta-data; we believe that 8-way replication is quite feasible given the “read-mostly” nature of such meta-data and the minimal space overhead (less than 1%) this entails.

The database log plays a salient role in the recoverability of the database, and its ability to make use of partial availability. It is therefore important for the log to be available under multiple failures. We believe that providing high availability for the log is indeed possible. Given that the size of the “active portion” of the log is determined by the length of the longest transaction factored by the concurrency in the workload, the portion of the log that needs to be kept highly available is quite reasonable. Modern storage arrays have large amounts of persistent RAM, which are obvious locations to place the log for high availability, perhaps replicating it across multiple NVRAM stores. This, in addition to normal on-disk storage of the log, can ensure that the log remains accessible in the face of multiple disk failures.

4.3.4 Access-Driven Diffusion

As stated above, with coarse-grained fragmentation, an entire table is placed within a single disk. If the table is large or is accessed frequently, this can have a performance impact since the parallelism that can be obtained across the disks is wasted. To remedy this, D-GRAID monitors accesses to the logical address space and tracks logical segments that are likely to benefit from parallelism. D-GRAID then creates an extra copy of those blocks and spreads them across the disks in the array, like a normal RAID would do. Thus, for blocks that are “hot”, D-GRAID regains the lost parallelism due to collocated layout, while still providing partial availability guarantees. Reads and writes are first sent to the diffused copy, with background updates being sent to the actual copy.

This technique underneath of a DBMS is essentially identical to that used underneath a file system.

4.3.5 Infallible Writes

Partial availability of data introduces interesting problems for the transaction and recovery mechanisms within a DBMS. For example, a transaction is often declared “committed” after it is reflected in the log. In a partially available system, after a crash, a redo for the transaction can fail if some pages are not available, which may seem to affect the durability semantics of transactions. However, this problem has already been considered and solved in ARIES [20], in the context of handling offline objects during *deferred restart*.

To ensure transaction durability, D-GRAID implements infallible writes, in which it guarantees that a write “always” succeeds. If a block to be written is destined for a dead disk, D-GRAID remaps it into a live disk and writes it (assuming that there is free space remaining on a live disk). This remapping prevents a new failure when flushing an already committed transaction to disk.

4.4 Evaluation

We evaluate the availability improvements and performance of D-GRAID through a prototype implementation; our D-GRAID prototype functions as a software RAID driver in the Linux 2.4 kernel, and operates underneath the Predator/Shore DBMS.

4.4.1 Availability Improvements

To evaluate availability improvements with D-GRAID, we use a D-GRAID array of 16 disks, and study the fraction of queries that the database serves successfully under an increasing number of disk failures. Since layout techniques in D-GRAID are complementary to existing RAID schemes such as parity or mirroring, we show D-GRAID Level 0 (*i.e.*, no redundancy for data) in our measurements, for simplicity. We mainly use microbenchmarks to analyze the availability provided by various layout techniques in D-GRAID.

A. Coarse-grained fragmentation

We first evaluate the availability improvements due to the coarse-grained fragmentation techniques in D-GRAID. Figure 1 presents the availability of scan, index lookup, and join queries for synthetic workloads under multiple disk failures. The percentage of such queries that complete successfully is reported.

The leftmost graph in Figure 1 shows the availability for scan queries. The database had 200 tables, each with 10,000 tuples. The workload is as follows: each query chooses a table at random and computes an average over a non-indexed attribute, thus requiring a scan of the entire table. As the graph shows, collocation of whole tables enables the database to be partially available, serving a proportional fraction of queries. In comparison, just one

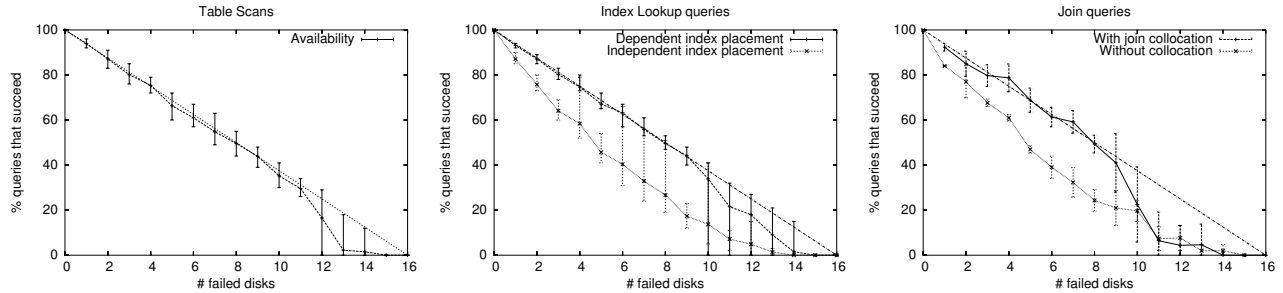


Figure 1: **Coarse-grained fragmentation.** The graphs show the availability degradation for scans, index lookups and joins under varying number of disk failures. A 16-disk D-GRAID array was used. The steeper fall in availability for higher number of failures is due to the limited (8-way) replication of metadata. The straight diagonal line depicts “ideal” linear degradation.

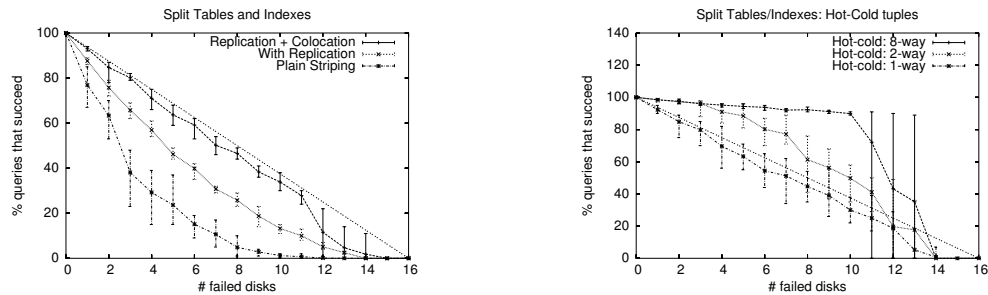


Figure 2: **Index Lookups under fine-grained fragmentation.** The graphs show the availability degradation for index lookup queries. The left graph considers a uniformly random workload, while the right graph considers a workload where a small set of tuples are very popular

failure in a traditional RAID-0 system results in complete unavailability. Note that if redundancy is maintained (*i.e.*, parity or mirroring), both D-GRAID and traditional RAID will tolerate up to one failure without any availability loss.

The middle graph in Figure 1 shows the availability for index lookup queries under a similar workload. We consider two different layouts; in both layouts, an entire “store” (*i.e.*, an index or a table) is collocated within one disk. In *independent index placement*, D-GRAID treats the index and table as independent stores and hence possibly allocates different disks for them, while with *dependent index placement*, D-GRAID carefully allocates the index on the same disk as the corresponding table. As can be seen, dependent placement leads to much better availability under failure.

Finally, to evaluate the benefits of join-group collocation, we use the following micro-benchmark: the database contains 100 pairs of tables, with joins always involving tables in the same pair. We then have join queries randomly select a pair and join the corresponding two tables. The rightmost graph in Figure 1 shows that by collocating joined tables, D-GRAID achieves higher availability.

B. Fine-grained fragmentation

We now evaluate the effectiveness of fine-grained fragmentation. We focus on the availability of index lookup

queries since they are the most interesting in this category. The workload we use for this study consists of index lookup queries on randomly chosen values of a primary key attribute in a single large table. We plot the fraction of queries that succeed under varying number of disk failures. The left graph in Figure 2 shows the results.

There are three layouts examined in this graph. The lowermost line shows availability under simple striping with just replication of system catalogs. As can be seen, the availability falls drastically under multiple failures due to loss of internal B-tree nodes. The middle line depicts the case where internal B-tree nodes are replicated aggressively; as can be expected, this achieves better availability. Finally, the third line shows the availability when data and index pages are collocated, in addition to internal B-tree replication. Together, these two techniques ensure near linear degradation of availability.

The right graph in Figure 2 considers a similar workload, but a small subset of tuples are much “hotter” compared to the others. Specifically, 5% of the tuples are accessed in 90% of the queries. Even under such a workload, simple replication and collocation provide near linear degradation in availability since hot pages are spread nearly uniformly across the disks. However, under such a hot-cold workload, D-GRAID can improve availability further by replicating data and index pages containing

	D-GRAID	RAID-0	Slowdown
Table Scan	7.97 s	6.74 s	18.1%
Index Lookup	51 ms	49.7 ms	2.7%
Bulk Load	186.61 s	176.14 s	5.9%
Table Insert	11.4 ms	11 ms	3.6%

Table 1: **Time Overheads of D-GRAID.** The table compares the performance of D-GRAID under fine-grained fragmentation, with default RAID-0 under various microbenchmarks. An array of 4 disks is used.

such hot tuples. The other two lines depict availability when such hot pages are replicated by factors of 2 and 8. Thus, when a small fraction of (read mostly) data is hot, D-GRAID utilizes that information to enhance availability through selective replication.

4.4.2 Performance overheads

We now evaluate the performance implications of fault-isolated layout in D-GRAID. For all experiments in this section, we use a 4-disk D-GRAID array comprised of 9.1 GB IBM UltraStar 9LZX disks with peak throughput of 20 MB/s. The database used has a single table of 500,000 records, each sized 110 bytes, with an index on the primary key.

A. Time and space overheads

We first explore the time and space overheads incurred by our D-GRAID prototype for tracking information about the database and laying out blocks to facilitate graceful degradation. Table 1 compares the performance of D-GRAID with fine-grained fragmentation to Linux software RAID 0 under various basic query workloads. The workloads examined are a scan of the entire table, an index lookup of a random key in the table, bulk load of the entire indexed table, and inserts into the indexed table. D-GRAID performs within 6% of RAID-0 for all workloads except scans. The poor performance in scans is due to a Predator anomaly, where the scan workload completely saturated the CPU (6.74 s for a 50 MB table across 4 disks). Thus, the extra CPU cycles required by D-GRAID impacts the scan performance by about 18%. This interference is because our prototype competes for resources with the host; in a hardware RAID system, such interference would not exist. Overall, we find that the overheads of D-GRAID are quite reasonable.

We also evaluated the space overheads due to aggressive metadata replication and found them to be minimal; the overhead scales with the number of tables, and even in a database with 10,000 tables, the overhead is only about 0.9% for 8-way replication of important data.

B. Access-driven Diffusion

We now evaluate the benefits of diffusing an extra copy of popular tables. Table 2 shows the time taken for a scan of

	Scan Time (s)
RAID-0	6.74
D-GRAID	15.69
D-GRAID + Diffusion	7.35

Table 2: **Diffusing Collocated Tables.** The table shows the scan performance on a 4-disk array under coarse-grained fragmentation.

the table described above, under coarse-grained fragmentation in D-GRAID. As can be seen, simple collocation leads to poor scan performance due to the lost parallelism. With the extra diffusion aimed at performance, D-GRAID performs much closer to default RAID-0.

4.5 Comparison

In our implementation of D-GRAID underneath a DBMS, we uncovered some fundamental challenges that were not present under a file system. First, the notion of semantically-related groups is more complex in a DBMS because of the various inter-relationships that exist across tables and indexes. In the file system case, whole files or whole directories were reasonable approximations of semantic groupings. In a DBMS, since the goal of D-GRAID is to enable serving as many higher level queries as possible, the notion of semantic grouping is *dynamic*, i.e., it depends on the query workload. Second, identifying “popular” data that needs to be aggressively replicated, is easier in file systems; standard system binaries and libraries were obvious targets, independent of the specific file system running above. However, in a DBMS, the set of popular tables varies with the DBMS and is often dependent on the query workload. Thus, effectively implementing D-GRAID underneath a DBMS requires slightly modifying the DBMS to record additional information. Finally, to ensure transaction durability, we implemented infallible writes for the version under the DBMS.

Comparing how well D-GRAID performs beneath a DBMS versus a file system we see many similarities. For example, both versions of D-GRAID successfully enable graceful degradation of availability; that is, both versions enable at least the expected number of processes or queries to complete successfully, given a fixed number of disk failures. In fact, both versions enable *more* than the expected number to complete when a subset of the data is especially popular. Similarly, both versions of D-GRAID do introduce some time overhead; interestingly, the slowdowns for our database version are generally lower than those for the file system version. Finally, both versions require access-driven diffusion to obtain acceptable performance.

5 Secure Delete with FADED

Our second case study is to implement FADED [28] underneath a DBMS. FADED is a semantically smart disk that detects deletes of records and tables at the DBMS level and securely overwrites (*i.e.*, *shreds*) the relevant data to make it irrecoverable. We extend previous work that implemented the same functionality for file systems [28].

5.1 Motivation

Deleting data such that recovery is impossible is important for file system security [4, 14]. Government regulations require guarantees on sensitive data being *forgotten*, and such requirements could become more important in databases [2]. Recent legislations on data retention, such as the Sarbanes-Oxley Act, have accentuated the importance of secure deletion.

Secure deletion of data in magnetic disks involves overwriting disk blocks with a sequence of writes with certain specific patterns to cancel out remnant magnetic effects due to past layers of data in the block. While early work indicated that as many as 32 overwrites per block are required for secure erase [14], recent work shows that two to three such overwrites suffice for modern disks [16].

Neither a file system nor a DBMS can ensure secure deletion when it functions on top of modern storage systems, which transparently perform various optimizations. For example, the storage system could buffer writes in NVRAM before writing them out to disk [34]. In the presence of NVRAM buffering, multiple overwrites done by the file system or DBMS may be collapsed into a single write to the physical disk, making the overwrites ineffective. Also, in the presence of block migration within the storage system [9], overwrites by the file system or DBMS will miss past copies.

Thus, secure deletion requires the low level information and control that the storage system has, and at the same time, higher level semantic information about the file system or DBMS to detect logical deletes. A semantically-smart disk system is thus an ideal locale to implement secure deletion.

5.2 Filesystem-Aware FADED

When running underneath a file system, FADED infers that a file is deleted by tracking writes to inodes, indirect blocks, and bitmap blocks and looking for changes. Due to the asynchronous nature of file systems, FADED is not able to guarantee that the current contents of a block belong to the deleted file and not to a newly allocated file (which should not be shredded). To ensure that it does not shred valid data, FADED uses *conservative overwrites* in which it shreds only an old version of a block before restoring the current contents of the block.

In previous work, we implemented FADED for three file systems: Linux ext2, Linux ext3, and Windows VFAT.

However, for FADED to work correctly, each file system had to be changed. For example, Linux ext2 was modified to ensure that data bitmap blocks are flushed whenever an indirect block is allocated or freed; Windows VFAT was changed to track a generation number for each file; finally, Linux ext3 was modified so that the list of modified data blocks are included in each transaction.

5.3 Database-Aware FADED

To implement FADED beneath a DBMS, the semantic disk must be able to identify and handle deletes for both entire tables as well as for individual records. We discuss these two cases in turn.

The simplest case for FADED is when a whole table is deleted. When a `drop table` command is issued, FADED must shred all blocks belonging to the table. FADED uses log snooping to identify log records that indicate freeing of extents from stores. In SHORE, a `free_ext_list` log record is written for every extent freed. Once FADED knows the list of freed blocks, it can issue secure overwrites to those pages. If the transaction aborts (thus undoing the deletes), the contents of the freed pages will be required; therefore, FADED pessimistically waits until the transaction is committed before performing any overwrites.

Handling record-level deletes in FADED is more challenging. When specific tuples are deleted (via the SQL `delete from` statement), specific byte ranges in the pages containing those tuples must be shredded. On a delete, a DBMS typically marks the relevant page “slot” free, and increments the free space count in the page. Since such freeing of slots is logged, FADED can learn of such record deletes by log snooping. However, FADED cannot shred the whole page because other records in the page could still be valid. Rather than read the current page from disk, we defer the shredding until FADED receives a write to the page reflecting the relevant delete. On receiving such a write, FADED shreds the entire page in the disk, and then writes the new data received. However, there are two complications with this basic technique.

The first complication is to identify the correct version of the page containing the deleted record. Assume that FADED observes a record delete d in page P , and waits for a subsequent write of P . When P is written, FADED needs to detect if the version written reflects d . The version could be stale if the DBMS wrote the page sometime before the delete, but the block was reordered by the disk scheduler and arrives later at the disk. This issue is similar to that of the file-system version of FADED; however, rather than use conservative overwrites, the database-aware version uses the WAL property of the DBMS to ensure correct operation. Specifically, database-aware FADED uses the *PageLSN* field in the page [20] to identify whether P reflects the delete. The *PageLSN* of a page tracks the sequence number of the

	Run time (s)	
	Workload I	Workload II
Default	52.0	66.0
FADED ₂	78.3	128.5
FADED ₄	91.0	160.0
FADED ₆	104.5	190.2

Table 3: **Overheads of secure deletion.** *This table shows the performance of FADED with 2, 4 and 6 overwrites, under two workloads. Workload I deletes contiguous records, while Workload II deletes records randomly across the table.*

latest log record describing a change in the page. Thus, FADED simply needs to compare the *PageLSN* to the LSN of the delete *d*.

The second complication is that the DBMS may not zero out bytes that belonged to deleted records; as a result, old data still remains in the page. Thus, when FADED observes the page write, it scans the page looking for free space and explicitly zeroes out the deleted byte ranges. Since the page could remain in the DBMS cache, all subsequent writes to the page must also be scanned and zeroed out appropriately.

5.4 Evaluation

We now briefly evaluate the cost of secure deletion in FADED through a prototype implementation. The prototype is implemented as a device driver in the Linux 2.4 kernel, and works underneath Predator [27].

We consider two workloads operating on a table with 500,000 110-byte records. In the first workload, we perform a `delete from` in such a way that all rows in the second half of the table are deleted (*i.e.*, the deleted pages are contiguous). In the second workload, the tuples to be deleted are selected in random.

Table 3 compares the default case without FADED to FADED using two, four, and six overwrite passes. As expected, secure deletion comes at a performance cost due to the extra disk I/O for the multiple passes of overwrites. Given that modern disks can effectively shred data with only two overwrites [16], we focus on *FADED*₂; in this case, performance is 50% to 95% slower. However, since such overhead is incurred only on deletes, and only sensitive data needs to be deleted in this manner, we believe the costs are reasonable in situations where the additional security is required.

5.5 Comparison

The primary difference between the two versions of FADED is that the database-aware version is able to leverage the transactional properties of the DBMS to definitively track whether a particular block should be shredded. As a result, while the file system version of FADED

required changes to the file system (with the exception of data journaled ext3), our implementation of FADED does not require any DBMS changes. However, our version does require detailed information about the on-disk page layout of the DBMS. Furthermore, the record-level granularity of deletes in a DBMS makes secure deletion more complex than in its file system counterpart.

Both versions of FADED incur some overhead, depending upon the workload and the number of overwrites. On our two delete-intensive database workloads, FADED was 50% or 95% slower with two overwrites. Similarly, for the two file system workloads, FADED was 51% to 280% slower with two overwrites (from Table 11 of [28]). In summary, the slowdown incurred by FADED depends more on the workload and the number of overwrites than on whether it is used by a DBMS or a file system.

6 Exclusive Caching with X-RAY

Our final case study is to implement X-RAY [3] underneath a DBMS. X-RAY is an exclusive caching mechanism for storage arrays that attempts to cache disk blocks which are *not* present in the higher-level buffer cache, thus providing the illusion of a single large LRU cache. Previous work has demonstrated that this approach performs very well when the buffer cache is maintained by a file system [3].

6.1 Motivation

Modern storage arrays possess large amounts of RAM for caching disk blocks. For instance, a high-end EMC storage array has up to 256 GB of main memory for caching. Typically, this cache is a second-level cache and the file system or a database system maintains its own buffer cache in the host main memory. Current caching mechanisms in storage arrays do not account for this; a block is placed in the array cache on a read, duplicating the same blocks cached above. Cache space is thus wasted due to inclusion. A better strategy would be for the contents of the buffer cache and the disk array cache to be *exclusive*.

Wong et al. [35] proposed to avoid cache inclusion by modifying the file system and the disk interface to support a SCSI “demote” command, which enables treating the disk array cache as a victim cache. For a database system, their approach would require the DBMS to inform the disk about evictions from its buffer pool. However, requiring an explicit change to the SCSI storage interface makes this scheme hard to deploy, since industry consensus is required for adopting such a change.

6.2 Filesystem-Aware X-RAY

X-RAY predicts the contents of the file system buffer pool and then chooses to cache only the most recent victims in its own cache; X-RAY requires no changes to the storage interface. X-RAY uses access time statistics (*i.e.*, which block was accessed and when) to perform its predictions;

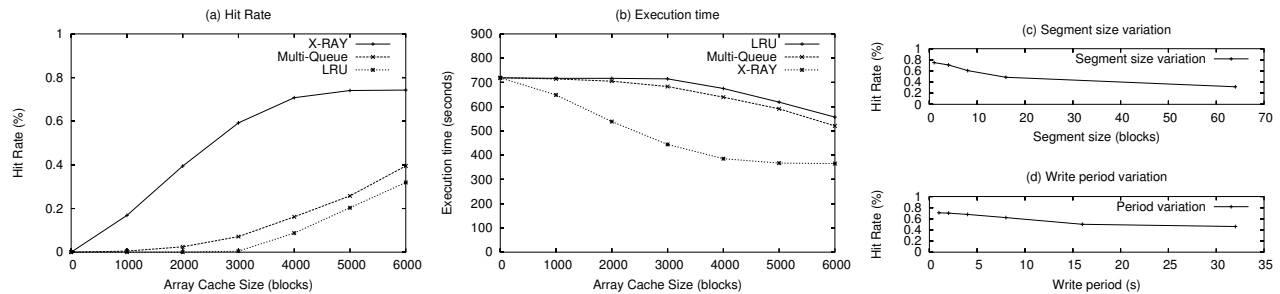


Figure 3: X-RAY Performance. The figure presents an evaluation of X-RAY under the TPC-C benchmark. The DBMS buffer cache was set to 6000 blocks for these studies. (a) The hit rate of X-RAY is compared to other caching mechanisms. The segment size is 4 blocks, and access information is written every 1 second. (b) The corresponding execution times are compared. The times are based on a buffer cache hit time of 20 μ s, a disk array cache hit time of 0.2 ms, a disk read time of 3 ms to 10 ms. (c) Hit-rate of X-RAY is measured for different segment sizes; the write period is kept at 1 second. (d) The write period is varied and the X-RAY hit rate is measured. The segment size is kept at 4 blocks.

for file systems such as Linux ext2, access statistics are recorded at the granularity of a file and are directly available in file inodes.

X-RAY uses these access statistics to maintain an ordered list of block numbers, from the LRU block to the MRU block. This is complicated by the fact that access statistics are tracked only a per-file basis. The ordered list is updated when X-RAY obtains new information, such as when the file system reads from disk (making the read block the most recently accessed) and when the file system writes an access time to disk. When a disk read arrives to a block *A*, X-RAY infers that *A* was evicted from the buffer cache some time in the past; it can also infer that any block *B* with an earlier access time was evicted as well (assuming an LRU policy). If the access time of block *A* is updated, but X-RAY did not observe a disk read for *A*, then X-RAY infers that block *A*, and all blocks with a later access time than *A*, are present in the buffer cache.

If the higher-level cache policy is LRU (which is the usual case), blocks close to the MRU end of the list are predicted to be in the file system buffer cache. The blocks near the LRU part of the list are considered the exclusive set; X-RAY caches the most recent blocks in the exclusive set, using extra internal array bandwidth or idle time between disk requests to read these blocks into the cache.

6.3 Database-Aware X-RAY

The database-aware version of X-RAY is very similar to the file system-aware version. The primary difference in creating a database-aware X-RAY occurs because a DBMS does not typically track access statistics. Although some database systems do maintain access statistics for administrative purposes (*e.g.*, AWR [21] for Oracle), these statistics are coarse in granularity and are written out only after long intervals.

Therefore, to implement database-aware X-RAY we must modify the database buffer manager to write out access statistics periodically. Specifically, each table or in-

dex is divided into fixed-sized *segments*, and the buffer manager periodically writes to disk the access time for segments accessed during the last period of time. X-RAY assumes that all blocks in the segment have been accessed when it sees that the access time statistic is updated. Thus, the accuracy with which X-RAY can predict the contents of the database cache is sensitive to both the size of each segment and the update interval. One advantage of explicitly adding this information is that we can tune the implementation by changing the size of the segment or the update interval. An alternative to adding this access information would be to modify the DBMS to directly report when it has evicted a block from its own cache, as in DEMOTE [35]. We believe that adding just access statistics is a better approach because the statistics are more general and can be used by semantic disks implementing other functionality (*e.g.*, D-GRAID [29]).

6.4 Evaluation

We evaluate the performance of our database-aware version of X-RAY with a simulation of both the database buffer cache and the disk array cache; the evaluation of the filesystem-aware X-RAY was performed using a simulation as well. The database buffer cache is maintained in LRU fashion; the DBMS periodically writes out access information at the granularity of one segment. The array cache is managed by X-RAY. We assume that X-RAY has sufficient internal bandwidth for its block reads.

We instrumented the buffer cache manager of the Postgres DBMS [24] to generate traces of page requests at the buffer cache level. We use Postgres because Predator does not have a programming API in Linux, which is required to implement TPC-C. We use an approximate implementation of the TPC-C benchmark for our evaluation (it adheres to the TPC-C specification [32] in its access pattern). A total of 5200 transactions are performed.

We evaluate the performance of X-RAY in terms of array cache hit rate and execution time. We compare X-

	Static				Dynamic				
	Catalog tables	Log record format	B-tree page format	Data page format	Transaction status	Block ownership	Block type	Block relationships	Access statistics
D-GRAID									
<i>basic</i>	x	x				x	x	x	
<i>+fine-grained frags</i>	x	x	x			x	x	x	
<i>+join-collocation</i>	x	x				x	x	x	x
FADED									
<i>basic</i>	x	x			x	x	x		
<i>+record-level delete</i>	x	x		x	x	x	x		
X-RAY									
<i>basic</i>	x	x				x			x

Table 4: **DBMS Information required for case studies.** *The table lists the static information that must be embedded into the semantic disk and the dynamic state that is automatically tracked by the disk.*

RAY to plain LRU and the Multi-Queue mechanism [36] designed for second level caches. We also explore sensitivity to segment size and access time update periodicity.

Figure 3a compares the hit rate of X-RAY with that of the other schemes and Figure 3b compares the corresponding execution times. The segment size is set to four blocks and access information is written out every second for this study. We see that X-RAY has much better hit rate than both LRU and Multi-Queue. This hit rate advantage extends to execution time despite the overhead of writing out the access information; X-RAY performs up to 75% better than LRU and up to 65% better than Multi-Queue.

Figure 3c evaluates the sensitivity of the X-RAY cache hit rate to segment size. As expected, the hit rate drops slightly with an increase in segment size. Figure 3d shows sensitivity to the access information update interval. We see that X-RAY can tolerate a reasonable delay (*e.g.*, about 5 seconds) when obtaining access updates.

6.5 Comparison

The file system and database versions of X-RAY are quite similar. To implement X-RAY, the semantic disk requires access statistics; that is, it must know which blocks are being accessed by the layer above. Although most file systems track and periodically write such statistics, a DBMS does not. Therefore, to use X-RAY, the DBMS must be modified to explicitly track access times for segments within each table. One advantage of explicitly adding this information is that one can tune the statistics more appropriately (*i.e.*, the size of segment and the update interval). Whether running beneath a file system or a database, X-RAY was found to substantially improve the array cache hit rate, relative to both LRU and Multi-Queue.

7 Information for Case Studies

In this section, we review the static and dynamic information required within a database-aware disk, given that this needed information depends upon the functionality that it is being implemented. The exact information required for variants of our three case studies is listed in Table 4.

Probably the biggest concern for database vendors is the static information that must be exported; for example, if a storage system understands the format of a particular catalog table, then the database vendor may be loathe to change its format. The amount of static information varies quite a bit across the case studies. While all of our case studies must know the format of catalog tables and log records, only D-GRAID with support for fine-grained fragmentation and FADED with record-level deletes need more detailed knowledge, such as the B-tree page format and the data page format, respectively.

The useful dynamic information also varies across case studies. The most fundamental piece of dynamic information is block ownership, as shown by the fact that it is required by every case study; block type is also a generally useful property, needed by both D-GRAID and FADED. The other pieces of dynamic information are not widespread. For example, only FADED needs to know precisely when a transaction has committed, since to be correct, it must be pessimistic in determining when to overwrite data; only D-GRAID needs to be able to associate blocks from a table with the blocks from the corresponding index, and vice versa. Finally, access correlation and access count statistics are needed by one of the D-GRAID variants to collocate related tables and to aggressively replicate “hot” data; the simple access time statistic is needed by X-RAY to predict the contents of the higher-level buffer cache.

8 Conclusions

“Today we [the database community] have this sort of simple-minded model that a disk is one arm on one platter and [it holds the whole database]. And in fact [what’s holding the database] is RAID arrays, it’s storage area networks, it’s all kinds of different architectures underneath that hood, and it’s all masked over by a logical volume manager written by operating system people who may or may not know anything about databases. Some of that transparency is really good because it makes us more productive and they just take care of the details. ... But on the other hand, optimizing the entire stack would be even better. So, we [in the two fields] need to talk, but on the other hand we want to accept some of the things that they’re willing to do for us.” [26].

-Pat Selinger

Semantic knowledge in the storage system enables powerful new functionality to be constructed. For example, the storage system can improve performance with

better caching [3], can improve reliability [29], and can provide additional security guarantees [28]. In this paper, we have shown that semantic storage technology can be deployed not only beneath commodity file systems, but beneath database management systems as well.

We have found that some different techniques are required to handle database systems. First, we investigated the impact of transactional semantics within the DBMS. In most cases, transactions simplify the work of a semantic disk. For example, log snooping enables the storage system to observe the operations performed by the DBMS and to definitively infer dynamic information without changing the DBMS. However, the storage system must also ensure that it does not interfere with the transactional semantics. For example, we found that infallible writes are useful to ensure transaction durability after some disks have failed. Second, we explored how the lack of access statistics within a DBMS complicates its interactions with a semantic disk. In this case, we found that it was helpful to slightly modify the database system to gather and relay simple statistics.

Acknowledgements

We would like to thank David Black for encouraging us to extend the semantic disks work to databases. We also thank David DeWitt, Jeff Naughton, Rajasekar Krishnamurthy and Vijayan Prabhakaran for their insightful comments on earlier drafts of this paper, and Jeniffer Beckham for pointing to us the Pat Selinger quote. Finally, we thank the anonymous reviewers for their thoughtful suggestions, many of which have greatly improved this paper.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0325267, IBM, Network Appliance, and EMC.

References

- [1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: programming model, algorithms and evaluation. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, California, October 1998.
- [2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *28th VLDB*, 2002.
- [3] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *ISCA '04*, 2004.
- [4] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *USENIX Security*, August 2001.
- [5] H. Boral and D. J. DeWitt. Database Machines: An Idea Whose Time has Passed? In *3rd International Workshop on Database Machines*, 1983.
- [6] J. Brown and S. Yamaguchi. Oracle's Hardware Assisted Resilient Data (H.A.R.D.). *Oracle Technical Bulletin (Note 158367.1)*, 2002.
- [7] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [8] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX '02*, pages 177–190, 2002.
- [9] EMC Corporation. Symmetrix Enterprise Information Storage Systems. <http://www.emc.com>, 2005.
- [10] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.
- [11] J. Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.
- [12] S. D. Gribble. Robustness in Complex Systems. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.
- [13] E. Grochowski. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. *Datatech*, September 1999.
- [14] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security*, July 1996.
- [15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *SIGMOD '97*, pages 171–182, 1997.
- [16] G. Hughes and T. Coughlin. Secure Erase of Disk Drive Data. *IDEMA Insight Magazine*, 2002.
- [17] K. Keeton. *Computer Architecture Support for Database Applications*. PhD thesis, University of California at Berkeley, 1999.
- [18] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the 10th ACM-SIGOPS European Workshop*, pages 93–100, Saint-Emilion, France, September 2002.
- [19] M. Carey et. al. Shoring up persistent applications. In *Proceedings of the 1994 ACM SIGMOD Conference*, 1994.
- [20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1):94–162, March 1992.
- [21] Oracle. The Self-managing Database: Automatic Performance Diagnosis. <https://www.oracleworld2003.com/published/40092/40092.doc>, 2003.
- [22] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, 1988.
- [23] D. A. Patterson. Availability and Maintainability >>> Performance: New Focus for a New Century. Key Note Lecture at FAST '02, 2002.
- [24] Postgres. The PostgreSQL Database. <http://www.postgresql.com>.
- [25] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *VLDB 24*, 1998.
- [26] P. Selinger and M. Winslett. Pat Selinger Speaks Out. *SIGMOD Record*, 32(4):93–103, December 2003.
- [27] P. Seshadri and M. Paskin. PREDATOR: An OR-DBMS with Enhanced Data Types. In *SIGMOD '97*, 1997.
- [28] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.
- [29] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST '04*, 2004.
- [30] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, 2003.
- [31] D. Slotnick. *Logic Per Track Devices*, volume 10, pages 291–296. Academic Press, 1970.
- [32] TPC-C. Transaction Processing Performance Council. <http://www.tpc.org/tpcc/>.
- [33] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.
- [34] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.
- [35] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *USENIX '02*, 2002.
- [36] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX '01*, pages 91–104, 2001.

Managing Prefetch Memory for Data-Intensive Online Servers*

Chuanpeng Li and Kai Shen

Department of Computer Science, University of Rochester

{cli, kshen}@cs.rochester.edu

Abstract

Data-intensive online servers may contain a significant amount of prefetched data in memory due to large-granularity I/O prefetching and high execution concurrency. Using a traditional access recency or frequency-based page reclamation policy, memory contention can cause a substantial number of prefetched pages to be prematurely evicted before being accessed. This paper presents a new memory management framework that handles prefetched (but not-yet-accessed) pages separately from the rest of the memory buffer cache. We examine three new heuristic policies when a victim page (among the prefetched pages) needs to be identified for eviction: 1) evict the last page of the longest prefetch stream; 2) evict the last page of the least recently accessed prefetch stream; and 3) evict the last page of the prefetch stream whose owner process has consumed the most amount of CPU since it last accessed the prefetch stream. These policies require no application changes or hints on their data access patterns.

We have implemented the proposed techniques in the Linux 2.6.10 kernel and conducted experiments based on microbenchmarks and two real application workloads (a trace-driven index searching server and the Apache Web server hosting media clips). Compared with access history-based policies, our memory management scheme can improve the server throughput of real workloads by 11–64% at high concurrency levels. Further, the proposed approach is 10–32% below an approximated optimal page reclamation policy that uses application-provided I/O access hints. The space overhead of our implementation is about 0.4% of the physical memory size.

1 Introduction

Emerging data-intensive online services access large disk-resident datasets while serving many clients simultaneously. Examples of such servers include Web-scale keyword search engines that support interactive search

on terabytes of indexed Web pages and Web servers hosting large multimedia files. For data-intensive online servers, the disk I/O performance and memory utilization efficiency dominate the overall system throughput when the dataset size far exceeds the available server memory. During concurrent execution, data access of one request handler can be frequently interrupted by other active request handlers in the server. Due to the high storage device seeking overhead, large-granularity I/O prefetching is often employed to reduce the seek frequency and thus decrease its overhead. At high execution concurrency, there can be many memory pages containing prefetched but not-yet-accessed data, which we call *prefetched pages* in short.

The prefetched pages are traditionally managed together with the rest of the memory buffer cache. Existing memory management methods generally fall into two categories.

- Application-assisted techniques [6, 16, 22, 23, 29] achieve efficient memory utilization with application-supplied information or hints on their I/O access patterns. However, the reliance on such information affects the applicability of these approaches and their relatively slow adoption in production operating systems is a reflection of this problem. Our objective in this work is to provide *transparent* memory management that does not require any explicit application assistance.
- Existing transparent memory management methods typically use access history-based page reclamations, such as Working-Set [8], LRU/LFU [19], or CLOCK [27]. Because prefetched pages do not have any access history, an access recency or frequency-based memory management scheme tends to evict them earlier than pages that have been accessed before. Although MRU-like schemes may benefit prefetched pages, they perform poorly with general application workloads and in practice they are only used for workloads with exclusively sequential data access pattern. Among prefetched pages themselves, the eviction order based on traditional reclamation policies would be actually First-In-First-Out, again due to the lack of any access his-

*This work was supported in part by NSF grants CCR-0306473, ITR/IIS-0312925, and an NSF CAREER Award CCF-0448413.

tory. Under such management, memory contention at high execution concurrency may result in premature eviction of prefetched pages before they are accessed, which we call *page thrashing*. It could severely degrade the server performance.

One reason for the lack of specific attention on managing the prefetched (but not-yet-accessed) memory pages is that these pages only constitute a small portion of the memory in normal systems. However, their presence is substantial for data-intensive online servers at high execution concurrency. Further, it was recently argued that more aggressive prefetching should be supported in modern operating systems due to emerging application needs and the disk I/O energy efficiency [21]. Aggressive prefetching would also contribute to more substantial presence of prefetched memory pages.

In this paper, we propose a new memory management framework that handles prefetched pages separately from other pages in the memory system. Such a separation protects against excessive eviction of prefetched pages and allows a customized page reclamation policy for them. We explore heuristic page reclamation policies that can identify the prefetched pages least likely to be accessed in the near future. In addition to the page reclamation policy, the prefetch memory management must also address the memory allocation between caching and prefetching. We employ a gradient descent-based approach that dynamically adjusts the memory allocation for prefetched pages in order to minimize the overall page miss rate in the system.

A number of earlier studies have investigated I/O prefetching and its memory management in an integrated fashion [5, 14, 22, 23]. These approaches can adjust the prefetching strategy depending on the memory cache content and therefore achieve better memory utilization. Our work in this paper is exclusively focused on the prefetch memory management with a given prefetching strategy. Combining our results with adaptive prefetching may further improve the overall system performance but it is beyond the scope of this work.

The rest of the paper is organized as follows. Section 2 discusses the characteristics of targeted data-intensive online servers and describes the existing OS support. Section 3 presents the design of our proposed prefetch memory management techniques and its implementation in the Linux 2.6.10 kernel. Section 4 provides the performance results based on microbenchmarks and two real application workloads. Section 5 describes the related work and Section 6 concludes the paper.

2 Targeted Applications and Existing OS Support

Our work focuses on online servers supporting highly concurrent workloads that access a large amount of disk-resident data. We further assume that our targeted workloads involve mostly read-only I/O. In these servers, each incoming request is serviced by a request handler which can be a thread in a multi-threaded server or a series of event handlers in an event-driven server. The request handler then repeatedly accesses disk data and consumes the CPU before completion. A request handler may block if the needed resource is unavailable. While request processing consumes both disk I/O and CPU resources, the overall server throughput is often dominated by the disk I/O performance and the memory utilization efficiency when the application data size far exceeds the available server memory. Figure 1 illustrates such an execution environment.

During concurrent execution, sequential data access of one request handler can be frequently interrupted by other active request handlers in the server. This may severely affect I/O efficiency due to long disk seek and rotational delays. Anticipatory disk I/O scheduling [11] alleviates this problem by temporarily idling the disk so that consecutive I/O requests that belong to the same request handler are serviced without interruption. However, anticipatory scheduling may not be effective when substantial think time exists between consecutive I/O requests. The anticipation may also be rendered ineffective when a request handler has to perform some interleaving synchronous I/O that does not exhibit strong locality. Such a situation arises when a request handler simultaneously accesses multiple data streams. For example, the index searching server needs to produce the intersection of multiple sequential keyword indexes when answering multi-keyword queries.

Improving the I/O efficiency can be accomplished by employing a large I/O prefetching depth when the application exhibits some sequential I/O access patterns [26]. A larger prefetching depth results in less frequent I/O switching, and consequently yields fewer disk seeks per time unit. In practice, the default Linux and FreeBSD may prefetch up to 128 KB and 256 KB respectively in advance during sequential file accesses. We recently proposed a competitive prefetching strategy that can achieve at least half the optimal I/O throughput when there is no memory contention [17]. This strategy specifies that the prefetching depth should be equal to the amount of data that can be sequentially transferred within a single I/O switching period. The competitive prefetching depth is around 500 KB for several modern IBM and Seagate disks that were measured.

I/O prefetching can create significant memory de-

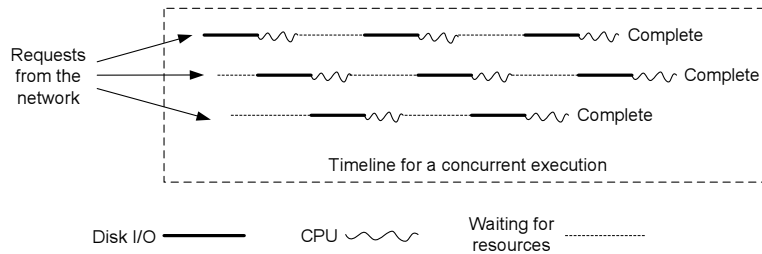


Figure 1: Concurrent application execution in a data-intensive online server.

mands. Such demands may result in severe contention at high concurrency levels, where many request handlers in the server are simultaneously competing for memory pages. Access recency or frequency-based replacement policies are misplaced for handling prefetched pages because they fail to give proper priority to pages that are prefetched but not yet accessed. We use the following two examples to illustrate the problems with access history-based replacement policies.

- *Priority between prefetched pages and accessed pages:* Assume page p_1 is prefetched and then accessed at t_1 while page p_2 is prefetched at t_2 and it has not yet been accessed. If the prefetching operation itself is not counted as an access, then p_2 will be evicted earlier under an access recency or frequency-based policy although it may be more useful in the near future. If the prefetching operation is counted as an access but t_1 occurs after t_2 , then p_2 will still be evicted earlier than p_1 .
- *Priority among prefetched pages:* Due to the lack of any access history, prefetched pages will follow a FIFO eviction order among themselves during memory contention. Assume page p_1 and p_2 are both prefetched but not yet accessed and p_1 is prefetched ahead of p_2 . Then p_1 will always be evicted earlier under the FIFO order even if p_2 's owner request handler (defined as the request handler that initiated p_2 's prefetching) has completed its task and exited from the server, an strong indication that p_2 would not be accessed in the near future.

Under poor memory management, many prefetched pages may be evicted before their owner request handlers get the chance to access them. Prematurely evicted pages will have to be fetched again and we call such a phenomenon prefetch page thrashing. In addition to wasting I/O bandwidth on makeup fetches, page thrashing further reduces the I/O efficiency. This is because page thrashing destroys sequential access sequence crucial for large-granularity prefetching and consequently many of the makeup fetches are inefficient small-granularity I/O operations.

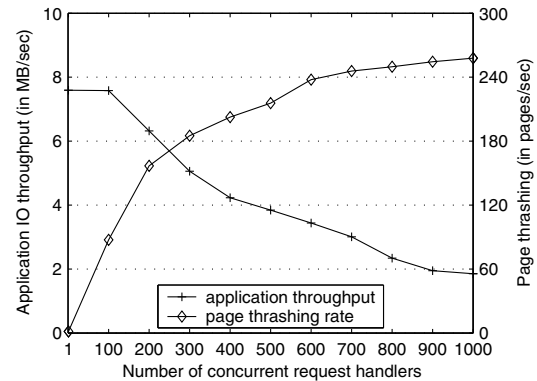


Figure 2: Application throughput and prefetch page thrashing rate of a trace-driven index searching server under the original Linux 2.6.10 kernel. A prefetch thrashing is counted when a prefetched page is evicted before the request handler that initiated the prefetching attempts to access it.

Measured performance We use the Linux kernel as an example to illustrate the performance of an access history-based memory management. Linux employs two LRU lists for memory management: an *active* list used to cache hot pages and an *inactive* list used to cache cold pages. Prefetched pages are initially attached to the tail of the inactive list. Frequently accessed pages are moved from the inactive list to the active list. When the available memory falls below a reclamation threshold, an aging algorithm moves pages from the active list to the inactive list and cold pages at the head of the inactive list are considered first for eviction.

Figure 2 shows the performance of a trace-driven index searching server under the Linux 2.6.10 kernel. This kernel is slightly modified to correct several performance anomalies during highly concurrent I/O [25]. The details about the benchmark and the experimental settings can be found in Section 4.1. We show the application throughput and page thrashing rate for up to 1000 concurrent request handlers in the server. At high execution concurrency, the I/O throughput drops significantly

as a result of high page thrashing rate. Particularly at the concurrency level of 1000, the I/O throughput degrades to about 25% of its peak performance while the prefetch page thrashing rate increases to about 258 pages/sec.

3 Prefetch Memory Management

We propose a new memory management framework with the aim to reduce prefetch page thrashing and improve the performance for data-intensive online servers. Our framework is based on a separate *prefetch cache* to manage prefetched but not-yet-accessed pages (shown in Figure 3). The basic concept of prefetch cache was used earlier by Papathanasiou and Scott [22] to manage the prefetched memory for energy-efficient bursty disk I/O. Prefetched pages are initially placed in the prefetch cache. When a page is referenced, it is moved from the prefetch cache to the normal memory buffer cache and is thereafter controlled by the kernel's default page replacement policy. At the presence of high memory pressure, some not-yet-accessed pages in the prefetch cache may be moved to the normal buffer cache (called *reclamation*) for possible eviction.

The separation of prefetched pages from the rest of the memory system protects against excessive eviction of prefetched pages and allows a customized page reclamation policy for them. However, like many of the other previous studies [6, 16, 23, 29], Papathanasiou and Scott's prefetch cache management [22] requires application-supplied information on their I/O access patterns. The reliance on such information affects the applicability of these approaches and our objective is to provide transparent memory management that does not require any application assistance. The design of our prefetch memory management addresses the reclamation order among prefetched pages (Section 3.1) and the partitioning between prefetched pages and the rest of the memory buffer cache (Section 3.2). Section 3.3 describes our implementation in the Linux 2.6.10 kernel.

3.1 Page Reclamation Policy

Previous studies pointed out that the offline optimal replacement rule for a prefetching system is that every prefetch should discard the page whose next reference is furthest in the future [4, 5]. Such a rule was introduced for minimizing the run time of a standalone application process. We adapt such a rule in the context of optimizing the throughput of data-intensive online servers, where a large number of request handlers execute concurrently and each request handler runs for a relatively short period of time. Since request handlers in an online server are independent from each other, the data prefetched by one request handler is unlikely to be ac-

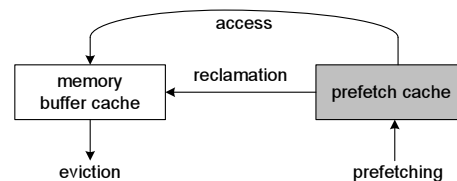


Figure 3: A separate prefetch cache for prefetched pages.

cessed by others in the near future. Our adapted *offline optimal replacement rules* are the following:

- A. If there exist prefetched pages that will not be accessed by their respective owner request handlers, discard one of these pages first.
- B. Otherwise, discard the page whose next reference is furthest in the future.

The optimal replacement rules can only be followed when the I/O access patterns for all request handlers are known. We explore heuristic page reclamation policies that can approximate the above rules without such information. We define a *prefetch stream* as a group of sequentially prefetched pages that have not yet been accessed. We examine the following online heuristics to identify a victim page for reclamation.

- #1. Discard any page whose owner request handler has completed its task and exited from the server.
- #2. Discard the last page from the longest prefetch stream.
- #3. Discard the last page from the prefetch stream whose last access is least recent. The recency can be measured by either the elapsed wall clock time or the CPU time of each stream's owner request handler. Since in most cases only a page's owner request handler would reference it, its CPU time may be a better indicator of how much opportunity of access it had in the past.

Heuristic #1 is designed to follow the offline replacement rule A while heuristic #2 approximates rule B. Heuristic #3 approximates both rules A and B since a prefetch stream that has not been referenced for a long time is not likely to be referenced soon and it may even not be referenced at all. Although heuristic #3 is based on access recency, it is different from traditional LRU in that it relies on the access recency of the prefetch stream it belongs to, not its own. Note that every page that was referenced before would have already been moved out of the prefetch cache.

Heuristic #1 can be combined with either heuristic #2 or #3 to form a page reclamation policy. In either case,

heuristic #1 should take precedence since it guarantees to discard a page that is unlikely to be accessed in the near future. When applying heuristic #3, we have two different ways to identify least recently used prefetch stream. Putting these together, we have three page reclamation policies when a victim page needs to be identified for reclamation:

- **Longest:** If there exist pages whose owner request handlers have exited from the server, discard one of them first. Otherwise, discard the last page from the longest prefetch stream.
- **Coldest:** If there exist pages whose owner request handlers have exited from the server, discard one of them first. Otherwise, discard the last page of the prefetch stream which has not been accessed for the longest time.
- **Coldest+:** If there exist pages whose owner request handlers have exited from the server, discard one of them first. Otherwise, discard the last page of the prefetch stream whose owner request handler has consumed the most amount of CPU since it last accessed the prefetch stream.

3.2 Memory Allocation for Prefetched Pages

The memory allocation between prefetched pages and the rest of the memory buffer cache can also affect the memory utilization efficiency. Too small a prefetch cache would not sufficiently reduce the prefetch page thrashing. Too large a prefetch cache, on the other hand, would result in many page misses on the normal memory buffer cache. To achieve high performance, we aim to minimize the combined prefetch miss rate and the cache miss rate. A *prefetch miss* is defined as a miss on a page which was prefetched and then evicted without being accessed while a *cache miss* is a miss on a page that has already been accessed at the time of last eviction.

Previous studies [14, 28] have proposed to use hit histograms to adaptively partition the memory among prefetching and caching or among multiple processes. In particular, based on Thiébaud *et al.*'s work [28], the minimal overall miss rate can be achieved when the current miss-rate derivative of the prefetch cache as a function of its allocated size is the same as the current miss-rate derivative of the normal memory buffer cache. This indicates an allocation point where there is no benefit of either increasing or decreasing the prefetch cache allocation. These solutions have not been implemented in practice and implementations suggested in these studies require significant overhead in maintaining page hit histograms. In order to allow a relatively light-weight implementation, we approximate Thiébaud *et al.*'s result using a *gradient descent*-based greedy algorithm [24]. Given

the curve of combined prefetch and cache miss rates as a function of the prefetch cache allocation, gradient descent takes steps proportional to the negative of the gradient at the current point and proceeds downhill toward the bottom.

We divide the runtime into epochs (or time windows) and the prefetch cache allocation is adjusted epoch-by-epoch based on gradient descending of the combined prefetch and cache miss rates. In the first epoch, we randomly increase or decrease the prefetch cache allocation by an *adjustment unit*. We calculate the change of combined miss rates in each subsequent epoch. This value is used to approximate the derivative of the combined miss rate. If the miss rate increases by a substantial margin, we reverse the adjustment of the previous epoch (*e.g.*, increase the prefetch cache allocation if the previous adjustment has decreased it). Otherwise, we further the adjustment made in the previous epoch (*e.g.*, increase the prefetch cache allocation further if the previous adjustment has increased it).

3.3 Implementation

We have implemented the proposed prefetch memory management framework in the Linux 2.6.10 kernel. In Linux, page reclamation is initiated when the number of free pages falls below a certain threshold. Pages are first moved from the active list to the inactive list and then the OS scans the inactive list to find candidate pages for eviction. In our implementation, when a page in the prefetch cache is referenced, it is moved to the inactive LRU list of the Linux memory buffer cache. Prefetch cache reclamation is triggered at each invocation of the global Linux page reclamation. If the prefetch cache size exceeds its allocation, prefetched pages will be selected for reclamation until the desired allocation is reached.

We initialize the prefetch cache allocation to be 25% of the total physical memory. Our gradient descent-based prefetch cache allocation requires the knowledge on several memory performance statistics (*e.g.*, the prefetch miss rate and the cache miss rate). In order to identify misses on recently evicted pages, the system maintains a bounded-length history of evicted pages along with flags that identify their status at the time of eviction (*e.g.*, whether they were prefetched but not-yet-accessed).

The prefetch cache data structure is organized as a list of stream descriptors. Each stream descriptor points to a list of ordered page descriptors, with newly prefetched page in the tail. A stream descriptor also maintains a link to its owner process (*i.e.*, the process that initiated the prefetching of pages in this stream) and the timestamp when the stream is last accessed. The timestamp is the wall clock time or the CPU run time of the stream's owner process depending on the adopted re-

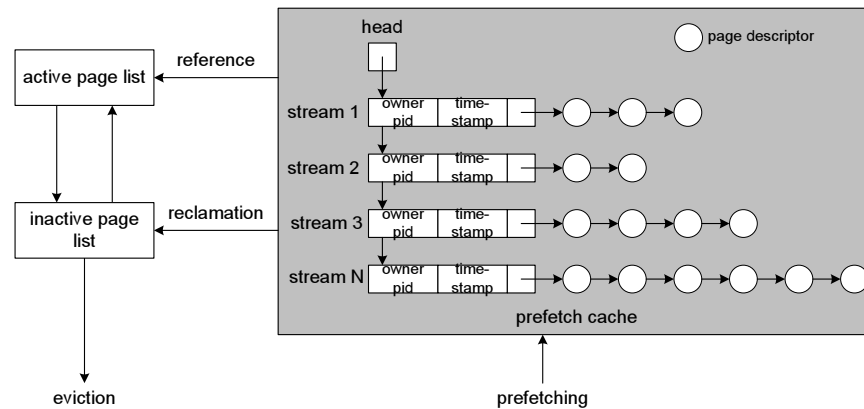


Figure 4: An illustration of the prefetch cache data structure and our implementation in Linux.

gency heuristic policy. Figure 4 provides an illustration of the prefetch cache data structure. In our current implementation, we assume a request handler is a thread (or process) in a multi-threaded (or multi-processed) server. Each prefetch stream can track the status of its owner process through the link in the stream descriptor. Due to its reliance on the process status, our current implementation cannot monitor the status of request handlers in event-driven servers that reuse each process for multiple request executions. In future implementation, we plan to associate each prefetch stream with its *open file* data structure directly. In this way, a file close operation will also cause the associated stream to be a reclamation candidate.

We assess the space overhead of our implementation. Since the number of active streams in the server is relatively small compared with the number of pages, the space consumption of stream headers is not a serious concern. The extra space overhead in each page includes two 4-byte pointers to form the stream page list. Therefore they incur 0.2% space overhead (8 bytes per 4 KB page). An additional space overhead is that used by the eviction history for identifying misses on recently evicted pages. We can limit this overhead by controlling the number of entries in the eviction history. A smaller history size may be adopted at the expense of the accuracy of the memory performance statistics. A 20-byte data structure is used to record information about each evicted page. When we limit the eviction history size to 40% of the total number of physical pages, the space overhead for the eviction history is about 0.2% of the total memory space.

4 Experimental Evaluation

We assess the effectiveness of our proposed prefetch memory management techniques on improving the per-

formance of data-intensive online servers. Experiments were conducted on servers each with dual 2.0 GHz Xeon processors, 2 GB memory, a 36.4 GB IBM 10 KRPM SCSI drive, and a 146 GB Seagate 10 KRPM SCSI drive. Note that we lower the memory size used in some experiments to better illustrate the memory contention. Each experiment involves a server and a load generation client. The client can adjust the number of simultaneous requests to control the server concurrency level.

The evaluation results are affected by several factors, including the I/O prefetching aggressiveness, server memory size, and the application dataset size. Our strategy is to first demonstrate the performance at a typical setting and then explicitly evaluate the impact of various factors. We perform experiments on several microbenchmarks and two real application workloads. Each experiment is run for 4 rounds and each round takes 120 seconds. The performance metric we use for all workloads is the I/O throughput observed at the application level. They are acquired by instrumenting the server applications with statistics-collection code. In addition to showing the server I/O throughput, we also provide some results on the prefetch page thrashing statistics. We summarize the evaluation results in the end of this section.

4.1 Evaluation Benchmarks

All microbenchmarks we use access a dataset of 6000 4 MB disk-resident files. On the arrival of each request, the server spawns a thread to process it. We explore the performance of four microbenchmarks with different I/O access patterns. They differ in the number of files accessed by each request handler (one to four) and the portion of each file accessed (the whole file, a random portion, or a 64 KB chunk). We use the following microbenchmarks in the evaluation:

- *One-Whole*: Each request handler randomly chooses a file and it repeatedly reads 64 KB data

Benchmark/workload	Whole-file access?	Streams per request	Memory Footprint	Total data size	Min-mean-max size of sequential access streams
Microbenchmark: One-Whole	Yes	Single	1 MB/request	24.0 GB	4 MB–4 MB–4 MB
Microbenchmark: One-Rand	No	Single	1 MB/request	24.0 GB	64 KB–2 MB–4 MB
Microbenchmark: Two-Rand	No	Multiple	1 MB/request	24.0 GB	64 KB–2 MB–4 MB
Microbenchmark: Four-64KB	No	N/A	1 MB/request	24.0 GB	N/A
Index searching	No	Multiple	Unknown	19.0 GB	Unknown
Apache hosting media clips	No	Single	Unknown	20.4 GB	24 KB–152 KB–1418 KB

Table 1: Benchmark statistics. The column “Whole-file access?” indicates whether a request handler would prefetch some data that it would never access. No such data exists if whole files are accessed by application request handlers. The column “Stream per request” indicates the number of prefetching streams each request handler initiates simultaneously. More prefetching streams per request handler would create higher memory contention.

blocks until the whole file is accessed.

- *One-Rand*: Each request handler randomly chooses a file and it repeatedly reads 64 KB data blocks from the file up to a random total size (evenly distributed between 64 KB and 4 MB).
- *Two-Rand*: Each request handler alternates reading 64 KB data blocks from two randomly chosen files. For each file, the request handler accesses the same random number of blocks. This workload emulates applications that simultaneously access multiple sequential data streams.
- *Four-64KB*: Each request handler randomly chooses four files and reads a 64 KB random data block from each file.

We also include two real application workloads in the evaluation:

- *Index searching*: We acquired an earlier prototype of the index searching server and a dataset from the Web search engine Ask Jeeves [3]. The dataset contains the search index for 12.6 million Web pages. It includes a 522 MB mapping file that maps MD5-encoded keywords to proper locations in the search index. The search index itself is approximately 18.5 GB, divided into 8 partitions. For each keyword in an input query, a binary search is first performed on the mapping file and then the search index is accessed following a sequential access pattern. Multiple prefetching streams on the search index are accessed for each multi-keyword query. The search query words in our test workload are based on a trace recorded at the Ask Jeeves online site in early 2002.
- *Apache hosting media clips*: We include the Apache Web server in our evaluation. Since our work focuses on data-intensive applications, we use a workload containing a set of media clips, following the file size and access distribution of the video/audio clips portion of the 1998 World Cup workload [2].

About 9% of files in the workload are large video clips while the rest are small audio clips. The overall file size range is 24 KB–1418 KB with an average of 152 KB. The total dataset size is 20.4 GB. During the tests, individual media files are chosen in the client requests according to a Zipf distribution. A random-size portion of each chosen file is accessed.

We summarize our benchmark statistics in table 1.

4.2 Microbenchmark Performance

We assess the effectiveness of the proposed techniques by comparing the server performance under the following five kernel versions:

- #1. *AccessHistory*: We use the original Linux 2.6.10 to represent kernels that manage the prefetched pages along with other memory pages using an access history-based LRU reclamation policy. Note that a more sophisticated access recency or frequency-based reclamation policy would not make much difference for managing prefetched pages. Although MRU-like schemes may benefit prefetched pages, they perform poorly with general application workloads and in practice they are only used for workloads with exclusively sequential data access pattern.
- #2. *PC-AccessHistory*: The original kernel with a prefetch cache whose allocation is dynamically maintained using our gradient-descent algorithm described in Section 3.2. Pages in the prefetch cache is reclaimed in FIFO-order, which is the effective reclamation order for prefetched pages under any access history-based reclamation policy.
- #3. *PC-Longest*: The original kernel with a prefetch cache managed by the Longest reclamation policy described in Section 3.1.

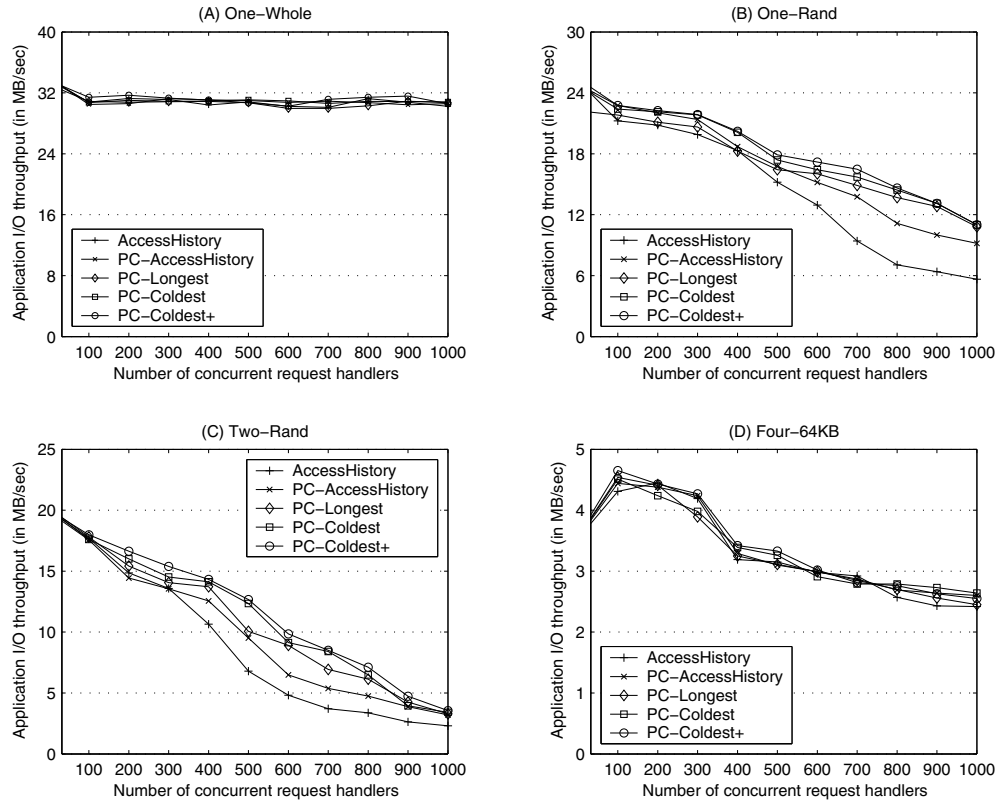


Figure 5: Microbenchmark I/O throughput at different concurrency levels. Our proposed prefetch memory management produces throughput improvement on One-Rand and Two-Rand.

#4. *PC-Coldest*: The original kernel with a prefetch cache managed by the Coldest reclamation policy described in Section 3.1.

#5. *PC-Coldest+*: The original kernel with a prefetch cache managed by the Coldest+ reclamation policy described in Section 3.1.

The original Linux 2.6.10 exhibits several performance anomalies when supporting data-intensive on-line servers [25]. They include a mis-management of prefetching during disk congestion and some lesser issues in the disk I/O scheduler. Details about these problems and some suggested fixes can be found in [25]. All experimental results in this paper are based on corrected kernels.

Figure 5 illustrates the I/O throughput of the four microbenchmarks under all concurrency levels. The results are measured with the maximum prefetching depth at 512 KB and the server memory size at 512 MB. Figure 5(A) shows consistently high performance attained by all memory management schemes for the first microbenchmark (One-Whole). This is because this microbenchmark accesses whole files and consequently all prefetched pages will be accessed. Further, for applica-

tions with strictly sequential access pattern, the anticipatory I/O scheduling allows each request handler to run without interruption. This results in a near-serial execution even at high concurrency levels, therefore little memory contention exists in the system.

Figure 5(B) shows the I/O throughput results for One-Rand. Since this microbenchmark does not access whole files, some prefetched pages will not be accessed. Results indicate that our proposed schemes can improve the performance of access history-based memory management at high concurrency. In particular, the improvement achieved by PC-Coldest+ is 8–97%.

Figure 5(C) shows the performance for Two-Rand. The anticipatory scheduling is not effective for this microbenchmark since a request handler in this case accesses two streams alternately. Results show that our proposed schemes can improve the performance of access history-based memory management at the concurrency level of 200 or higher. For instance, the improvement of PC-Coldest+ is 34% and two-fold at the concurrency levels of 400 and 500 respectively. This improvement is attributed to two factors: 1) the difference between PC-Coldest+ and PC-AccessHistory is attributed to the better page reclamation order inside the

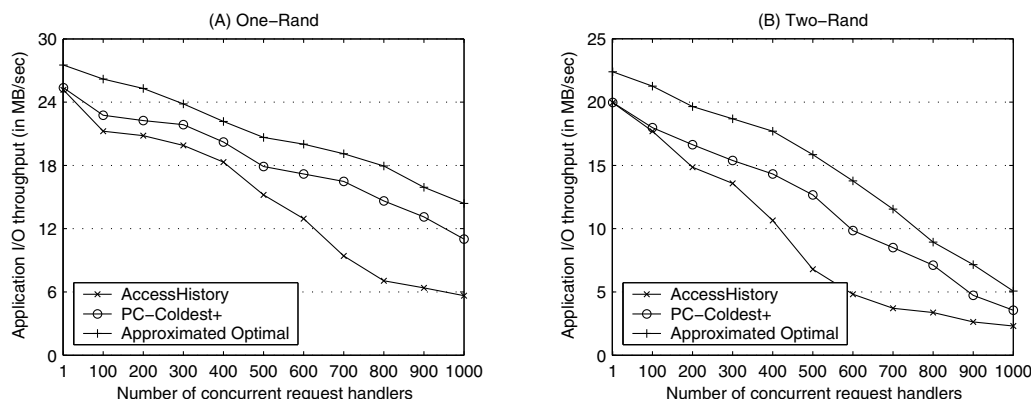


Figure 6: Comparison among AccessHistory, Coldest+, and an approximated optimal policy. The throughput of Coldest+ is 10–32% lower than that of the approximated optimal while it is 8–97% higher than that of AccessHistory in high concurrency.

prefetch cache; 2) the improvement of PC-AccessHistory over AccessHistory is due to the separation of prefetched pages from the rest of the memory buffer cache. Finally, we observe that the performance of PC-Coldest and PC-Longest are not as good as that of PC-Coldest+.

Figure 5(D) illustrates the performance of the random-access microbenchmark. We observe that all kernels perform similarly at all concurrency levels. Since prefetched pages in this microbenchmark are mostly not used, early eviction would not hurt the server performance. On the other hand, our memory partitioning scheme can quickly converge to small prefetch cache allocations through greedy adjustments and thus would not waste space on keeping prefetched pages.

Note that despite the substantial improvement on Two-Rand and One-Rand, our proposed prefetch memory management cannot eliminate memory contention and the resulted performance degradation at high concurrency. However, this is neither intended nor achievable. Consider a server with 1000 concurrent threads each prefetches up to 512 KB. Just the pages pinned for outstanding I/O operations alone occupies up to 512 MB memory space in this server.

To assess the remaining room for improvement, we approximate the offline optimal policy through direct application assistance. In our approximation, applications provide exact hints about their data access pattern through an augmented `open()` system call. With the hints, the approximated optimal policy can easily identify those prefetched but unneeded pages and evict them first (following the offline optimal replacement rule \mathcal{A} in Section 3.1). The approximated optimal policy does not accurately embrace offline optimal replacement rule \mathcal{B} , because there is no way to determine the access order among pages from different processes without the

process scheduling knowledge. For the eviction order among pages that would be accessed by their owner request handlers, the approximated optimal policy follows that of Coldest+. Figure 6 shows that the performance of Coldest+ approach is 10–32% lower than the approximated optimal in high concurrency with the two tested microbenchmarks.

4.3 Performance of Real Applications

We examine the performance of two real application workloads. Figure 7 shows the I/O throughput of the index searching server at various concurrency levels. The results are measured with the maximum prefetching depth at 512 KB and the server memory size at 512 MB. All schemes perform similarly at low concurrency. We notice that the I/O throughput initially increases as the concurrency level climbs up. This is mainly due to lower average seek distance when the disk scheduler can choose from more concurrent requests for seek reduction. At high concurrency levels (100 and above), the PC-Coldest+ scheme outperforms access history-based management by 11–64%. Again, this improvement is attributed to two factors. The performance difference between PC-AccessHistory and AccessHistory (about 10%) is due to separated management of prefetched pages while the rest is attributed to better reclamation order in the prefetch cache.

In order to better understand the performance results, we examine the page thrashing rate in the server. Figure 8 shows the result for the index searching server. We observe that the page thrashing rate increases as the server concurrency level goes up. Particularly at the concurrency of 1000, the page thrashing rates are around 258, 238, 217, 207 and 200 pages/request for the five schemes respectively. We observe that the difference

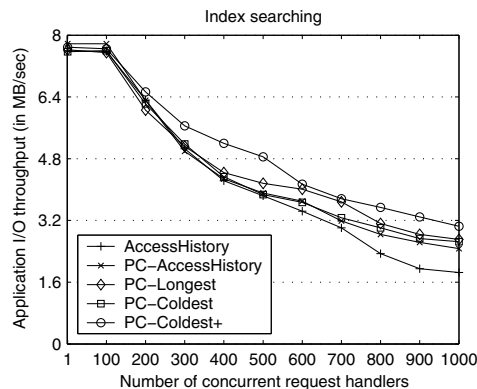


Figure 7: Throughput of the index searching server.

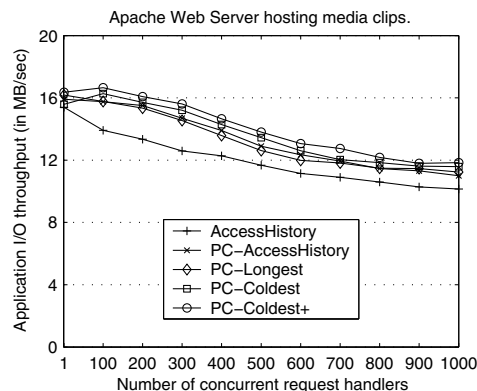


Figure 9: Throughput of the Apache Web server hosting media clips.

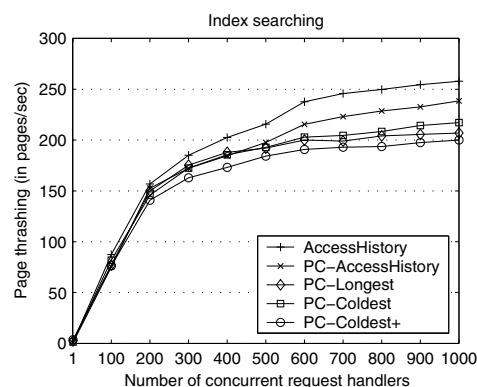


Figure 8: Prefetch page thrashing rate for the index searching server.

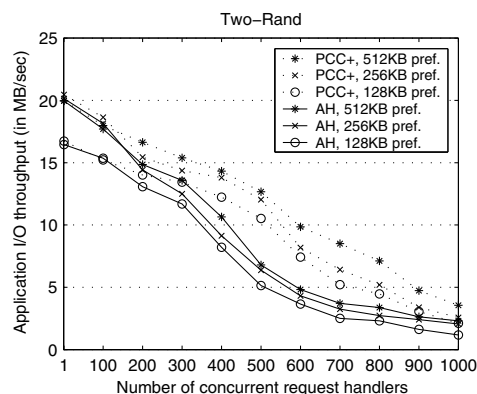


Figure 10: Performance impact of the maximum prefetching depth for the microbenchmark Two-Rand. “PCC+” stands for PC-Coldest+ and “AH” stands for AccessHistory.

in page thrashing rates across the four schemes matches well with that of the throughput performance in Figure 7, indicating that the page thrashing is a main factor in the server performance degradation at high concurrency. Even under the improved management, the page thrashing rate still increases in high concurrency levels for the same reason we described earlier for the microbenchmarks.

Figure 9 shows the application I/O throughput of the Apache Web server hosting media clips. The results are measured with the maximum prefetching depth at 512 KB and the server memory size at 256 MB. The four prefetch cache-based approaches perform better than access history-based approach in all concurrency levels. This improvement is due to separated management of prefetched pages. The difference among the four prefetch cache-based approaches is slight, but PC-Coldest+ persistently performs better than others. This is because of its effective reclamation management of prefetched pages.

4.4 Impact of Factors

In this section, we explore the performance impact of the I/O prefetching aggressiveness, server memory size, and the application dataset size. We use a microbenchmark in this evaluation because of its flexibility in adjusting the workload parameters. When we evaluate the impact of one factor, we set the other factors to default values: 512 KB maximum prefetching depth, 512 MB for the server memory size, and 24 GB for the application dataset size. We only show the performance of PC-Coldest+ and AccessHistory in the results.

Figure 10 shows the performance of the microbenchmark Two-Rand with different maximum prefetching depths: 128 KB, 256 KB, and 512 KB. Various maximum prefetching depths are achieved by adjusting the appropriate parameter in the OS kernel. We observe that our proposed prefetch memory management performs

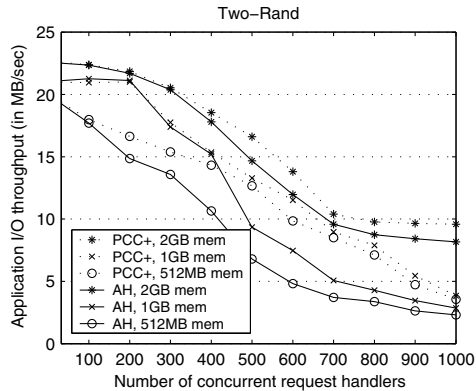


Figure 11: Performance impact of the server memory size for the microbenchmark Two-Rand. “PCC+” stands for PC-Coldest+ and “AH” stands for AccessHistory.

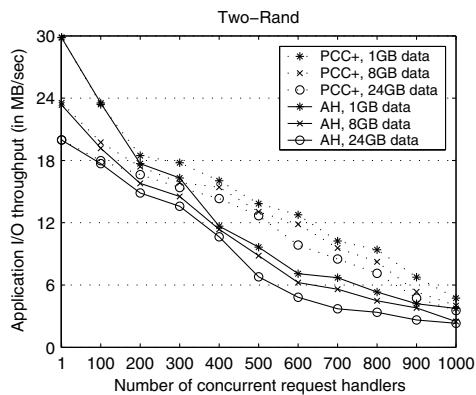


Figure 12: Performance impact of the workload data size for the microbenchmark Two-Rand. “PCC+” stands for PC-Coldest+ and “AH” stands for AccessHistory.

consistently better than AccessHistory at all configurations. The improvement tends to be more substantial for servers with higher prefetching depths due to higher memory contention (and therefore more room to improve).

Figure 11 shows the performance of the microbenchmark at different server memory sizes: 2 GB, 1 GB, and 512 MB. Our proposed prefetch memory management performs consistently better than AccessHistory at all memory sizes. The improvement tends to be less substantial for servers with large memory due to lower memory contention (and therefore less room to improve).

Figure 12 illustrates the performance of the microbenchmark at different workload data sizes: 1 GB, 8 GB, and 24 GB. Intuitively better performance is correlated with smaller workload data sizes. However, the change on the absolute performance does not significantly affect the performance advantage of our strategy

over access history-based memory management.

4.5 Summary of Results

- Compared with access history-based memory management, our proposed prefetch memory management scheme (PC-Coldest+) can improve the performance of real workloads by 11–64% at high execution concurrency. The performance improvement can be attributed to two factors: the separated management of prefetched pages and enhanced reclamation policies for them.
- At high concurrency, the performance of PC-Coldest+ is 10–32% below an approximated optimal page reclamation policy that uses application-provided I/O access hints.
- The microbenchmark results suggest that our scheme does not provide performance enhancement for applications that follow strictly sequential access pattern to access whole files. This is because the anticipatory I/O scheduler executes request handlers serially for these applications, thus eliminating memory contention even at high concurrency. In addition, our scheme does not provide performance enhancement for applications with only small-size random data accesses since most prefetched pages will not be accessed for these applications (and thus their reclamation order does not matter).
- The performance benefit of the proposed prefetch memory management may be affected by various system parameters. In general, the benefit is higher for systems with more memory contention. Additionally, our scheme does not degrade the server performance on all the configurations that we have tested.

5 Related Work

Concurrency management. The subject of highly-concurrent online servers has been investigated by several researchers. Pai *et al.* showed that the HTTP server workload with moderate disk I/O activities can be efficiently managed by incorporating asynchronous I/O or helper threads into an event-driven HTTP server [20]. A later study by Welsh *et al.* further improved the performance by balancing the load of multiple event-driven stages in an HTTP request handler [34]. The more recent Capriccio work provided a user-level thread package that can scale to support hundreds of thousands of threads [32]. These studies mostly targeted CPU-intensive workloads with light disk I/O activities (*e.g.*, the typical Web server workload and application-level

packet routing). They did not directly address the memory contention problem for data-intensive online servers at high concurrency.

Admission control. The concurrency level of an online server can be controlled by employing a fixed-size process/thread pool and a request waiting queue. QoS-oriented admission control strategies [18, 30, 31, 33] can adjust the server concurrency according to a desired performance level. Controlling the execution concurrency may mitigate the memory contention caused by I/O prefetching. However, these strategies may not always identify the optimal concurrency threshold when the server load fluctuates. Additionally, keeping the concurrency level too low may reduce the server resource utilization efficiency (due to longer average disk seek distance) and lower the server responsiveness (due to higher probability for short requests to be blocked by long requests). Our work complements the concurrency control management by inducing slower performance degradation when the server load increases and thus making it less critical to choose the optimal concurrency threshold.

Memory replacement. A large body of previous work has explored efficient memory buffer page replacement policies for data-intensive applications, such as Working-Set [8], LRU/LFU [19], CLOCK [27], 2Q [13], Unified Buffer Management [15], and LIRS [12]. All these memory replacement strategies are based on the page reference history such as reference recency or frequency. In data-intensive online servers, a large number of prefetched but not-yet-accessed pages may emerge due to aggressive I/O prefetching and high execution concurrency. Due to a lack of any access history, these pages may not be properly managed by access recency or frequency-based replacement policies.

Memory management for Caching and Prefetching. Cao *et al.* proposed a two-level page replacement scheme that allows applications to control their own cache replacement while the kernel controls the allocation of cache space among processes [6]. Patterson *et al.* examined memory management based on application-disclosed hints on application I/O access pattern [23]. Such hints were used to construct a cost-benefit model for deciding the prefetching and caching strategy. Kimbrel *et al.* explored the performance of application-assisted memory management for multi-disk systems [16]. Tomkins *et al.* further expanded the study for multi-programmed execution of several processes [29]. Hand studied application-assisted paging techniques with the goal of performance isolation among multiple concurrent processes [10]. Anastasiadis *et al.* explored an application-level block reordering technique that can reduce server disk traffic when large content files are shared by concurrent clients [1]. Memory man-

agement with application assistance or application-level information can achieve high (or optimal) performance. However, the reliance on such assistance may affect the applicability of these approaches and the goal of our work is to provide transparent memory management that does not require any application assistance.

Chang *et al.* studied the automatic generation of I/O access hints through OS-supported speculative execution [7, 9]. The purpose of their work is to improve the prefetching accuracy. They do not address the management of prefetched memory pages during memory contention.

Several researchers have addressed the problem of memory allocation between prefetched and cached pages. Many such studies required application assistance or hints [5, 16, 23, 29]. Among those that did not require such assistance, Thiébaud *et al.* [28] and Kaplan *et al.* [14] employed hit histogram-based cost-benefit models to dynamically control the prefetch memory allocation with the goal of minimizing the overall page fault rate. Their solutions require expensive tracking of page hits and they have not been implemented in practice. In this work, we employ a greedy partitioning algorithm that requires much less state maintenance and thus is easier to implement.

6 Conclusion

This paper presents the design and implementation of a prefetch memory management scheme supporting data-intensive online servers. Our main contribution is the proposal of novel page reclamation policies for prefetched but not-yet-accessed pages. Previously proposed page reclamation policies are principally based on the page access history, which are not well suited for pages that have no such history. Additionally, we employ a gradient descent-based greedy algorithm that dynamically adjusts the memory allocation for prefetched pages. Our work is guided by previous results on optimal memory partitioning while we focus on a design with low implementation overhead. We have implemented the proposed techniques in the Linux 2.6.10 kernel and conducted experiments using microbenchmarks and two real application workloads. Results suggest that our approach can substantially reduce prefetch page thrashing and improve application performance at high concurrency levels.

Acknowledgments The implementation of the statistics for evicted pages (mentioned in section 3.3) is based on a prototype by Athanasios Papathanasiou. We also would like to thank Athanasios Papathanasiou, Michael Scott, and the anonymous reviewers for their valuable

comments that greatly helped to improve this work.

References

- [1] S. V. Anastasiadis, R. G. Wickremesinghe, and J. S. Chase. Circus: Opportunistic Block Reordering for Scalable Content Servers. In *Proc. of the Third USENIX Conf. on File and Storage Technologies (FAST'04)*, pages 201–212, San Francisco, CA, March 2004.
- [2] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35, HP Laboratories Palo Alto, 1999.
- [3] Ask Jeeves Search. <http://www.ask.com>.
- [4] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, 1966.
- [5] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the ACM SIGMETRICS*, pages 188–197, Ottawa, Canada, June 1995.
- [6] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-Controlled File Caching. In *Proc. of the First USENIX Symp. on Operating Systems Design and Implementation (OSDI'94)*, Monterey, CA, November 1994.
- [7] F. Chang and G. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proc. of the Third USENIX Symp. on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, LA, February 1999.
- [8] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11:323–333, May 1968.
- [9] K. Fraser and F. Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *Proc. of the USENIX Annual Technical Conference*, San Antonio, TX, June 2003.
- [10] S. M. Hand. Self-paging in the Nemesis Operating System. In *Proc. of the Third USENIX Symp. on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, LA, February 1999.
- [11] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP'01)*, pages 117 – 130, Banff, Canada, October 2001.
- [12] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proc. of the ACM SIGMETRICS*, pages 31–42, Marina Del Rey, CA, June 2002.
- [13] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of the 20th VLDB Conference*, pages 439–450, Santiago, Chile, September 1994.
- [14] S. F. Kaplan, L. A. McGeoch, and M. F. Cole. Adaptive Caching for Demand Prepaging. In *Proc. of the Third Int'l Symp. on Memory Management*, pages 114–126, Berlin, Germany, June 2002.
- [15] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References. In *Proc. of the 4th USENIX Symp. on Operating Systems Design and Implementation (OSDI'00)*, San Diego, CA, October 2000.
- [16] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proc. of the Second USENIX Symp. on Operating Systems Design and Implementation (OSDI'96)*, Seattle, WA, October 1996.
- [17] C. Li, A. Papathanasiou, and K. Shen. Competitive Prefetching for Data-Intensive Online Servers. In *Proc. of the First Workshop on Operating System and Architectural Support for the on demand IT Infrastructure*, Boston, MA, October 2004.
- [18] K. Li and S. Jamin. A Measurement-Based Admission-Controlled Web Server. In *Proc. of the IEEE INFOCOM*, pages 651–659, Tel-Aviv, Israel, March 2000.
- [19] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems J.*, 9(2):78–117, 1970.
- [20] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [21] A. Papathanasiou and M. Scott. Aggressive Prefetching: An Idea Whose Time Has Come. In *Proc. of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, June 2005.

- [22] A. E. Papathanasiou and M. L. Scott. Energy Efficient Prefetching and Caching. In *Proc. of the USENIX Annual Technical Conference*, Boston, MA, June 2004.
- [23] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, pages 79–95, Copper Mountain Resort, CO, December 1995.
- [24] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.
- [25] K. Shen, M. Zhong, and C. Li. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proc. of the 4th USENIX Conf. on File and Storage Technologies (FAST'05)*, San Francisco, CA, December 2005.
- [26] E. Shriver, C. Small, and K. Smith. Why Does File System Prefetching Work ? In *Proc. of the USENIX Annual Technical Conference*, Monterey, CA, June 1999.
- [27] A. J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.
- [28] D. Thiébaud, H. S. Stone, and J. L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers*, 41(6):665–676, June 1992.
- [29] A. Tomkins, R. H. Patterson, and G. A. Gibson. Informed Multi-Process Prefetching and Caching. In *Proc. of the ACM SIGMETRICS*, pages 100–114, Seattle, WA, June 1997.
- [30] T. Voigt and P. Gunningberg. Dealing with Memory-Intensive Web Requests. Technical Report 2001-010, Uppsala University, May 2001.
- [31] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *Proc. of the USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [32] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP'03)*, pages 268–281, Bolton Landing, NY, October 2003.
- [33] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *Proc. of the 4th USENIX Symp. on Internet Technologies and Systems*, Seattle, WA, March 2003.
- [34] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP'01)*, pages 230–243, Banff, Canada, October 2001.

A Scalable and High Performance Software iSCSI Implementation

Abhijeet Joglekar, Michael E. Kounavis, and Frank L. Berry
Intel Research and Development, Hillsboro, OR, 97124, USA

Abstract

In this paper we present two novel techniques for improving the performance of the Internet Small Computer Systems Interface (iSCSI) protocol, which is the basis for IP-based networked block storage today. We demonstrate that by making a few modifications to an existing iSCSI implementation, it is possible to increase the iSCSI protocol processing throughput from 1.4 Gbps to 3.6 Gbps. Our solution scales with the CPU clock speed and can be easily implemented in software using any general purpose processor without requiring specialized iSCSI protocol processing hardware.

To gain an in-depth understanding of the processing costs associated with an iSCSI protocol implementation, we built an iSCSI fast path in a user-level sandbox environment. We discovered that the generation of Cyclic Redundancy Codes (CRCs) which is required for data integrity, and the data copy operations which are required for the interaction between iSCSI and TCP represent the main bottlenecks in iSCSI protocol processing. We propose two optimizations to iSCSI implementations to address these bottlenecks. Our first optimization is on the way CRCs are being calculated. We replace the industry standard algorithm proposed by Prof. Dilip Sarwate with ‘Slicing-by-8’ (SB8), a new algorithm capable of ideally reading arbitrarily large amounts of data at a time while keeping its memory requirement at reasonable level. Our second optimization is on the way iSCSI interacts with the TCP layer. We interleave the compute-intensive data integrity checks with the memory access-intensive data copy operations to benefit from cache effects and hardware pipeline parallelism.

1. Introduction

Networked block storage technologies are likely to play a major role in the development of next generation data centers. In this paper we address the problem of efficiently implementing networked block storage systems focusing on systems that operate on top of the TCP/IP protocol stack. We analyze the performance of the iSCSI protocol [30], which is the basis for IP-based networked block storage today and suggest ways to improve its performance. There are two primary reasons why we believe IP-based networked block

storage is important. First, such type of storage enables efficient remote backup and recovery operations on top of large-scale and geographically distributed networks. Second, using the same IP-based technology in both storage and regular communication networks makes network management easier and less expensive since there is only one type of network to manage. More elaborate discussions on networked storage are presented in [28, 33].

Commercial iSCSI solutions have been designed thus far using TCP/IP offload engines (TOEs) or iSCSI host bus adapters (HBAs). These systems offload either the TCP/IP protocol stack or both the TCP/IP and the iSCSI protocols into specialized hardware units. In this paper we follow an alternative approach to offloading by focusing on a software-only iSCSI implementation. The reason why we focus on a software iSCSI implementation is because such implementation scales better with the CPU clock speed and the number of processing units available and can be easily realized using general purpose processors without specialized iSCSI protocol processing hardware. Our work is also motivated by earlier studies that have demonstrated that accessing protocol offload engines may become a bottleneck for some protocol processing workloads. For example, Sarkar et al [27] compare a software iSCSI stack with two industry standard solutions, a TOE and an HBA, operating at 1 Gbps. Their paper shows that while current generation hardware solutions do achieve better throughput-utilization efficiency as compared to software for large block sizes, accessing the hardware offload engines becomes a bottleneck for small block sizes.

The contributions of this paper can be summarized as follows: First, through measurements and simulations performed on a sandbox implementation of iSCSI, we quantify the processing costs of each of the protocol components including data structure manipulation, CRC generation, and data copies. We identify the CRC generation and data copies as the primary bottleneck in iSCSI processing. Second we replace the industry-standard CRC generation algorithm developed by Prof. Dilip Sarwate [29] with a new ‘Slicing-by-8’ (SB8) algorithm, capable of ideally reading arbitrarily large amounts of data at a time while keeping its memory requirement at reasonable level. A third contribution of our paper is a novel way to implement the interaction between the iSCSI and TCP layers. We interleave the compute-intensive data integrity checks with the

memory access-intensive data copy operations to benefit from cache effects and hardware pipeline parallelism. This optimization was inspired by the idea of integrated copy-checksum as first suggested by Clark et al [6]. We demonstrate that these two novel implementation techniques can increase the processing throughput of our implementation from 1.4 Gbps to 3.6 Gbps. These optimizations correspond to a small number of changes in the source code of a software iSCSI implementation. Our work relies on the acceleration of TCP on the CPU which is a well researched problem [4, 5, 13, 26].

The paper is organized as follows. In Section 2 we provide an overview of the iSCSI protocol, and describe typical receive and transmit fast paths in an iSCSI initiator stack. The information presented in this section is essential so as the reader can understand our optimizations. For more information on iSCSI, the reader can look at [30]. In section 3, we propose two optimizations that address the two primary bottlenecks in an iSCSI implementation - the CRC generation process and the data copies. In Section 4, we describe our sandbox iSCSI implementation, and our measurement and simulation methodology. In Section 5 we evaluate our approach and discuss our results. In Section 6 we present related work in the area and, finally, in Section 7 we provide some concluding remarks.

2. Overview of iSCSI processing

2.1 The Protocol

The iSCSI protocol maps the SCSI client-server protocol onto a TCP/IP interconnect. Initiators (clients) on a SCSI interface issue commands to a SCSI target (server) in order to request the transfer of data to or from I/O devices. The iSCSI protocol encapsulates these SCSI commands and the corresponding data into iSCSI Protocol Data Units (PDUs) and transmits them over a TCP connection. An iSCSI PDU includes a Basic Header Segment (BHS), followed by one or more Additional Header Segments (AHS). Additional header segments are followed by a data segment. Headers and data are protected separately by a digest based on the CRC32c standard [30].

An iSCSI session has two phases. It starts with a 'login' phase during which the initiator and target negotiate the parameters for the rest of the session. Then, a 'full feature' phase is used for sending SCSI commands and data. Based on the parameters negotiated during the login phase, an iSCSI session can use multiple TCP connections multiplexed over one or more physical interfaces, enable data integrity checks over PDUs, and even incorporate different levels of

error recovery. iSCSI sessions are typically long-lived. The login phase represents only a small part of the overall protocol processing load. Because of this reason we have decided to investigate optimizations on the 'full feature' phase of the protocol only. Figure 1 depicts a typical layered protocol stack on an initiator.

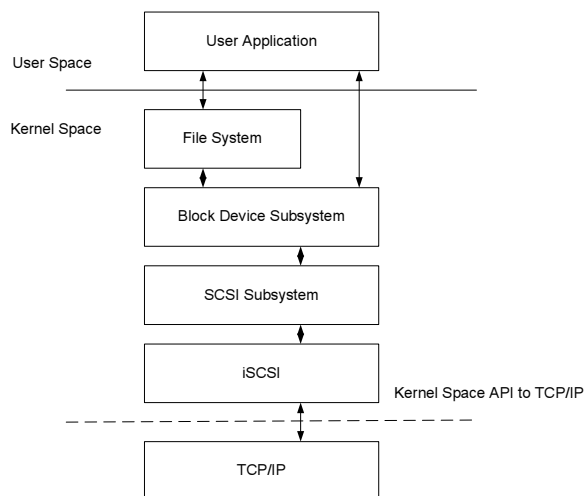


Figure 1: Typical initiator protocol stack

User-level applications issue 'read' and 'write' system calls that are serviced by the SCSI subsystem through the file system and block device layers. A SCSI 'upper' layer creates a SCSI Command Descriptor Block (CDB) and passes it to a SCSI 'lower' layer. The SCSI lower layer is the iSCSI driver for an IP interconnect. This iSCSI driver uses the kernel interface to the TCP/IP stack to transmit SCSI commands and data to a target. Each SCSI read command is transmitted as an iSCSI command PDU to a target. A target then services the read command by encapsulating SCSI data into one or multiple iSCSI data PDUs and by sending them to the initiator. The data PDUs are eventually followed by a status PDU from the target signaling the completion of the read operation. SCSI write commands are similarly implemented by first sending a SCSI write command to a target followed by a pre-negotiated number of unsolicited bytes. The target paces the flow of data from the initiator by issuing flow-control messages based on the availability of target buffers. As in the case of the read command, the end of the data transfer is indicated by the transmission of a status PDU from the target to the initiator.

2.2 iSCSI Read Processing

Figure 2 shows the processing of an incoming data PDU (also called 'data-in' PDU) in an initiator stack performing SCSI read operations.

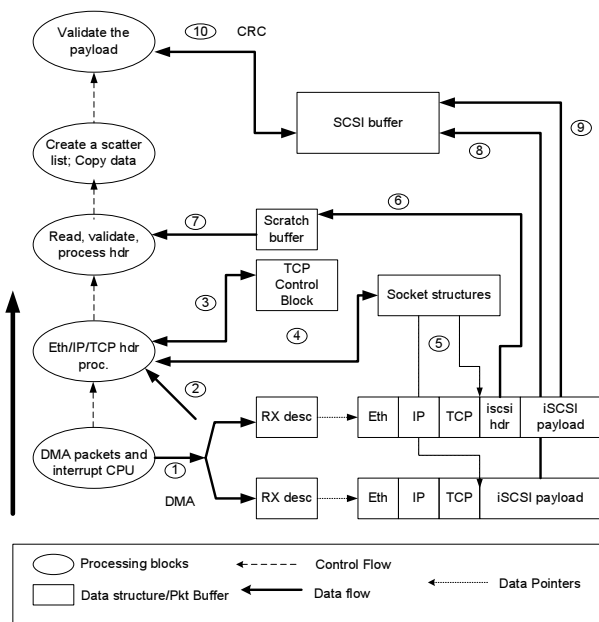


Figure 2: Incoming data PDU processing for reads

An iSCSI PDU, which has a default size of 8KB, can span multiple TCP segments. As the Network Interface Card (NIC) receives these segments, it places them using the Direct Memory Access (DMA) technique into a number of NIC buffers and interrupts the CPU (step 1 in the figure). The device driver and stack then use the Rx descriptors and the TCP Control block in order to perform TCP/IP processing and to strip off the Eth/IP/TCP headers (steps 2 and 3). The segment payloads are then queued into socket descriptor structures (steps 4 and 5). So far, steps 1-5 describe processing associated at the TCP/IP layer and below.

Steps 6-10 describe processing associated with the iSCSI layer. The iSCSI layer first reads the iSCSI header from the socket layer into an internal scratch buffer (step 6). The header consists of a fixed 48 byte basic header, a header CRC, and in some cases additional header bytes. It then computes a CRC over the header and validates this CRC value by comparing it with the CRC attached to the header. If the CRC value is valid, the iSCSI layer processes the header and identifies the incoming iSCSI PDU as a data PDU (step 7). A tag value included in the header is used for identifying the SCSI buffer where the data PDU should be placed. Based on the length of the PDU and its associated offset, both of which are indicated in the iSCSI header, the iSCSI layer creates a scatter-list pointing to the SCSI buffer. Then, the iSCSI layer passes the scatter list to the socket layer which copies the iSCSI PDU payload from the TCP segments into the SCSI buffer (steps 8 and 9). Finally, the data CRC is computed and validated over the entire PDU payload

(step 10). This is the last step in the processing of an incoming PDU.

2.3 iSCSI Write Processing

Figure 3 shows the handling of an outgoing data PDU (also called 'data-out' PDU) in an initiator stack performing SCSI writes to the target.

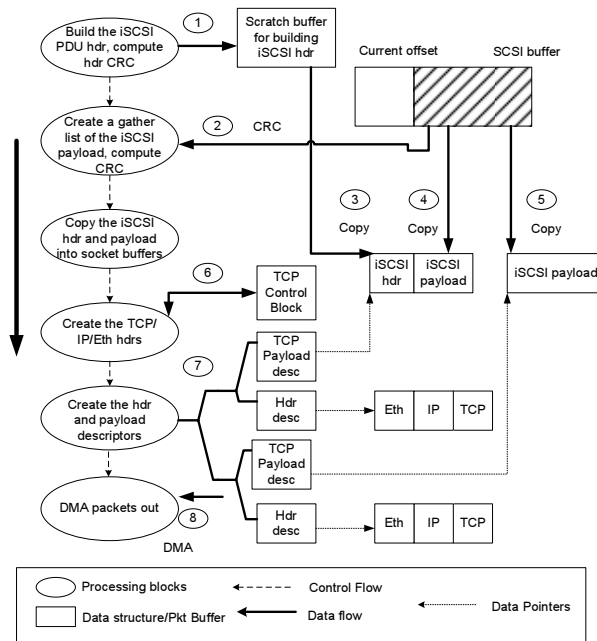


Figure 3: Outgoing data PDU processing for writes

The iSCSI protocol layer at the initiator maintains a pointer into the SCSI buffer for the next data PDU to be sent to the target. In reply to flow control (i.e., R2T) PDUs that are received from the target, the iSCSI layer transmits the data PDU. It first constructs the iSCSI header that describes the data PDU, computes a CRC value on the header, and attaches this CRC value to the header (step 1). The iSCSI layer then builds a gather list that describes the header, the header CRC, and the payload of the iSCSI PDU. It computes a data CRC on the payload and attaches it to the gather list (step 2). Finally, it uses the kernel interface to the TCP/IP stack to send the PDU to the target.

Based on the TCP/IP stack implementation, the data is either directly transmitted from the SCSI buffer to the NIC or undergoes a copy to temporary socket buffers, as shown in steps 3, 4, and 5. The TCP/IP stack then creates the transport, network, and link headers associated with each of the TCP segments that hold a portion of the iSCSI PDU. It also creates header and payload descriptors that point to the packet headers and the TCP payload respectively (steps 6 and 7). Finally,

the stack signals the NIC to send the packets out onto the wire via DMA (step 8).

3. Software Optimizations

One can expect that the CRC generation process and data copies are the most time consuming parts of iSCSI processing. CRC generation is costly because it requires several logical operations to be performed on a byte-by-byte basis for each byte of data. Data copies are costly because accessing off-chip memory units typically requires several hundreds of clock cycles to complete. In sections 4 and 5, we describe our sandbox implementation and processing profile of the iSCSI stack. The performance of our sandbox implementation was measured on a 1.7 GHz Intel® Pentium® M processor with a 400 MHz Front Side Bus (FSB), and a single channel DDR-266 memory subsystem. In summary, we found that the generation of CRC32c codes indeed represents the most time consuming component of the stack operating at a rate of 6.5 cycles per byte, followed by data copy operations that operate at a rate of 2.2 cycles per byte. In this section, we describe how we address these bottlenecks using a new CRC generation algorithm and a technique to interleave data copies with the CRC generation process.

3.1 The CRC generation process

Cyclic redundancy codes (CRC) are used for detecting the corruption of digital content during its production, transmission, processing or storage. CRC algorithms treat each bit stream as a binary polynomial $B(x)$ and calculate the remainder $R(x)$ from the division of $B(x)$ with a standard ‘generator’ polynomial $G(x)$. The binary words corresponding to $R(x)$ are transmitted together with the bit stream associated with $B(x)$. The length of $R(x)$ in bits is equal to the length of $G(x)$ minus one. At the receiver side, CRC algorithms verify that $R(x)$ is the correct remainder. Long division is performed using modulo-2 arithmetic. Additions and subtractions in module-2 arithmetic are ‘carry-less’ as illustrated in Table 1. In this way additions and subtractions are equal to the exclusive OR (XOR) logical operation.

$0+0 = 0-0 = 0$
$0+1 = 0-1 = 1$
$1+0 = 1-0 = 1$
$1+1 = 1-1 = 0$

Table 1: Modulo-2 arithmetic

Figure 4 illustrates a long division example. In the example, the divisor is equal to ‘11011’ whereas the dividend is equal to ‘1000111011000’. The long

division process begins by placing the 5 bits of the divisor below the 5 most significant bits of the dividend. The next step in the long division process is to find how many times the divisor ‘11011’ ‘goes’ into the 5 most significant bits of the dividend ‘10001’. In ordinary arithmetic 11011 goes zero times into 10001 because the second number is smaller than the first. In modulo-2 arithmetic, however, the number 11011 goes exactly one time into 10001. To decide how many times a binary number goes into another in modulo-2 arithmetic, a check is being made on the most significant bits of the two numbers. If both are equal to ‘1’ and the numbers have the same length, then the first number goes exactly one time into the second number, otherwise zero times. Next, the divisor 11011 is subtracted from the most significant bits of the dividend 10001 by performing an XOR logical operation. The next bit of the dividend, which is ‘1’, is then marked and appended to the remainder ‘1010’. The process is repeated until all the bits of the dividend are marked. The remainder that results from such long division process is the CRC value.

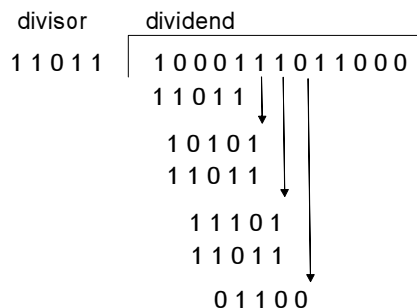


Figure 4: Long division using modulo-2 arithmetic

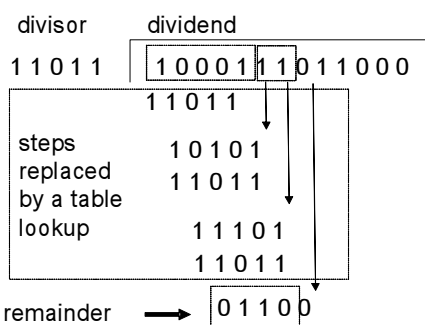


Figure 5: Accelerating the long division using table lookups

The long division process is a compute-intensive operation because it requires in the worst case one shift operation and one XOR logical operation for every bit of a bit stream. Most software-based CRC generation algorithms, however, perform the long division quicker than the bit-by-bit marking process described above.

One commonly used technique for accelerating the long division process is to pre-compute the current remainder that results from a group of bits and place the result in a table. Before the beginning of the long division process all possible remainders which result from groups of bits are pre-computed and placed into a lookup table. In this way, several long division steps can be replaced by a single table lookup step.

The main idea behind this technique is shown in Figure 5. In the example of Figure 5, the remainder '0110', which is formed in the third step of the long division process is a function of the five most significant bits of the dividend '10001' and the next two bits '11'. Since these bits are known, the remainder 0110 can be calculated in advance. As a result, 3 long division steps can be replaced by a single table lookup. Additional table lookups can further replace subsequent long division steps. To avoid using large tables, table-driven CRC acceleration algorithms typically read no more than 8 bits at a time.

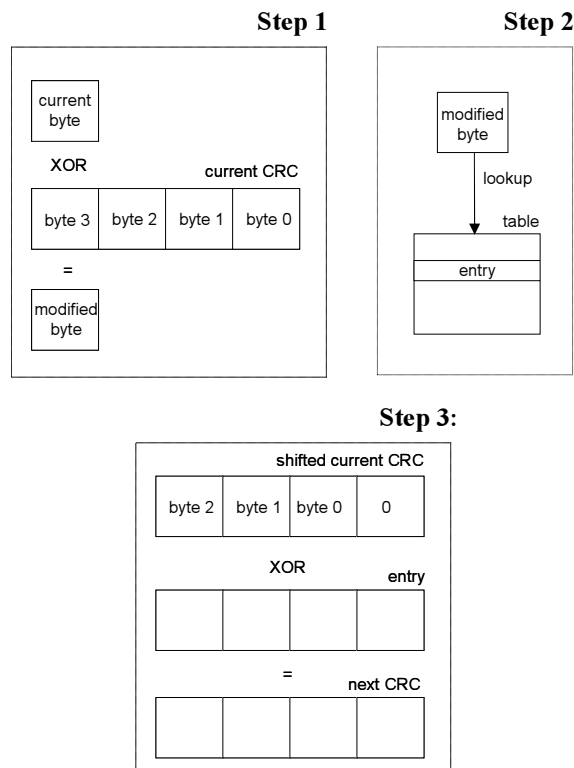


Figure 6: The Sarwate algorithm

The most representative table-driven CRC generation algorithm used today is the algorithm proposed by Dilip V. Sarwate, shown in Figure 6. The length of the CRC value generated by the Sarwate algorithm is 32 bits. The Sarwate algorithm is more complicated than the straightforward lookup process of Figure 5 because the

amount of bits read at a time (8 bits) is smaller than the degree of the generator polynomial.

Initially, the CRC value is set to a given number which depends on the standard implemented (e.g., this number is 0xFFFFFFFF for CRC32c). For every byte of an input stream the algorithm performs the following steps: First, the algorithm performs an XOR operation between the most significant byte of the current CRC value and the byte from the stream which is read (Step 1). The 8-bit number which is produced by this XOR operation is used as an index for accessing a 256 entry table (Step 2). The lookup table used by the Sarwate algorithm stores the remainders from the division of all possible 8-bit numbers shifted by 32 bits to the left with the generator polynomial. The value returned from the table lookup is then XOR-ed with the 24 least significant bits of the current CRC value, shifted by 8 bit positions to the left (Step 3). The result from this last XOR operation is the CRC value used in the next iteration of the algorithm's main loop. The iteration stops when all bits of the input stream have been taken into account. Detailed justification and proof of correctness of the Sarwate algorithm is beyond the scope of this paper. The reader can learn more about the Sarwate algorithm in [29].

3.2 Optimizing the CRC generation process

The Sarwate algorithm was designed at a time when most computer architectures supported XOR operations between 8 bit quantities. Since then, computer architecture technology has progressed to the point where arithmetic operations can be performed efficiently between 32 or 64 bit quantities. In addition modern computer architectures comprise large on-chip cache memory units which can be accessed in a few clock cycle time. We believe that such advances call for re-examination of the mathematical principles behind software-based CRC generation.

The main disadvantage of existing table-driven CRC generation algorithms is their memory space requirement when reading a large number of bits at a time. For example, to achieve acceleration by reading 32 bits at a time, table-driven algorithms require storing pre-computed remainders in a table of $2^{32} = 4G$ entries. To solve this problem we propose a new algorithm that slices the CRC value produced in every iteration as well as the data bits read into small terms. These terms are used as indexes for performing lookups on different tables in parallel. The tables differ between each other and are constructed in a different manner than Sarwate as explained in detail below. In this way, our algorithm is capable of reading 64 bits at a time, as opposed to 8, while keeping its memory space requirement to 8KB.

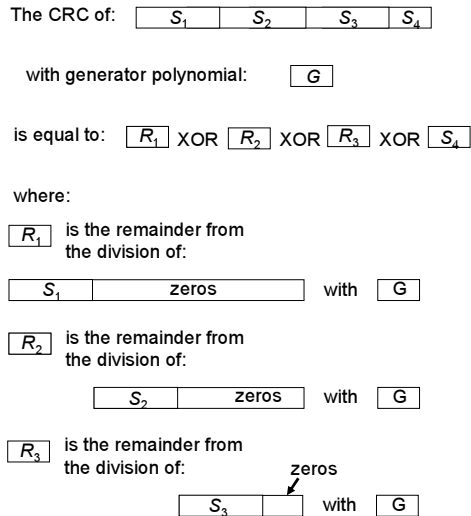


Figure 7: The Bit slicing principle

CRC is a linear code. This means that $CRC(A+B) = CRC(A) + CRC(B)$. The linearity of CRC derives from the arithmetic used (modulo 2). The design of our algorithm is based on two principles associated with the linearity of CRC, namely a ‘bit slicing’ and a ‘bit replacement’ principle. The bit slicing principle is shown in Figure 7. According to this principle, if a binary number is sliced into two or more constituent terms the CRC value associated with the binary number can be calculated as a function of the CRC values of its constituent terms. As it is shown in the figure, the CRC of the number consisting of slices S_1 , S_2 , S_3 , and S_4 is the result of an XOR operation between values R_1 , R_2 , R_3 , and S_4 . Value R_1 is the CRC of the original number if all slices but S_1 are replaced with zeros. Values R_2 and R_3 are defined in a similar manner. The bit slicing principle is important because it allows us to compute the CRC of a potentially large number as a function of the CRCs of smaller terms. Thus, the bit slicing principle can potentially solve the memory explosion problem associated with existing table-driven algorithms.

The bit replacement principle is shown in Figure 8. According to this principle, an arbitrarily long prefix of a bit stream can be replaced by an appropriately selected binary number, without changing the CRC value of the stream. The binary number used for replacing a prefix is the remainder from the division of the prefix with the CRC generator polynomial. In the example of Figure 8, the U_1 prefix of the binary number $[U_1:U_2]$ can be replaced by the remainder R_1 from the division of U_1 with G . It is this bit replacement principle which we take advantage of in the design of our algorithm in order to read 64 bits at a time.

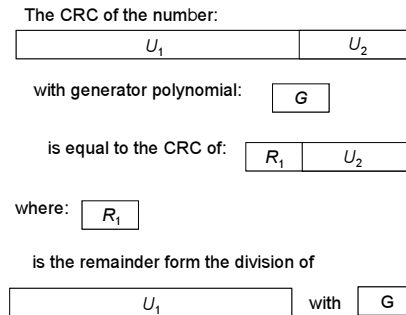


Figure 8: The Bit replacement principle

Our algorithm, called ‘Slicing-by-8’ (SB8), is illustrated in Figure 9. As in the case of the Sarwate algorithm, the initial CRC value of a stream is set to a given number which depends on the standard implemented (e.g., this number is 0xFFFFFFFF for CRC32c). For every 64-bit chunk of an input stream the algorithm performs the following steps: First, the algorithm performs an XOR operation between the 32 most significant bits of the chunk and the current CRC value (Step 1). The 64-bit value produced from this XOR operation is then sliced into 8 slices of equal length (Step 2). Each slice is used for accessing a separate lookup table (Step 3). The lookup tables used by the algorithm store the remainders from the division of all possible 8-bit numbers shifted by a variable number of bits to the left with the generator polynomial. The offset values used for calculating the table entries begin with 32 for Table 1 and increase by 8 for every table. The values returned from all table lookups are XOR-ed to one another producing the CRC value used in the next iteration of the algorithm’s main loop.

The benefit from slicing comes from the fact that modern processor architectures comprise large cache units. These cache units are capable of storing moderate size tables (e.g., 8KB tables as required by the Slicing-by-8 algorithm) but not sufficient for storing tables associated with significantly larger strides (e.g., 16GB tables associated with 32-bit strides). If tables are stored in an external memory unit, the latency associated with accessing these tables may be significantly higher than when tables are stored in a cache unit. For example, a DRAM memory access requires several hundreds of clock cycles to complete by a Pentium® M processor, whereas an access to a first level cache memory unit requires less than five clock cycles to complete. The processing cost associated with slicing is typically insignificant when compared to the cost of accessing off-chip memory units.

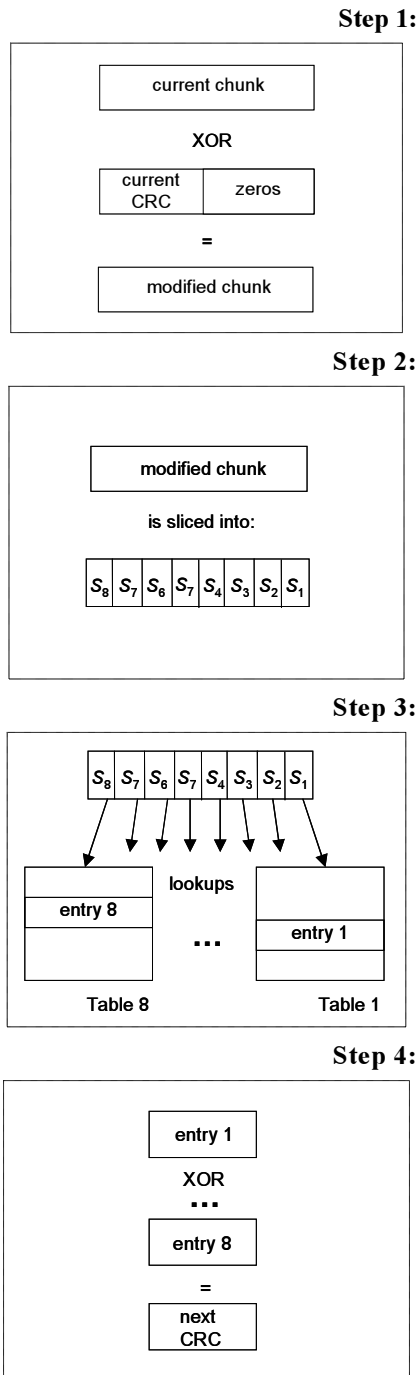


Figure 9: The Slicing-by-8 algorithm

Slicing is also important because it reduces the number of operations performed for each byte of an input stream when compared to Sarwate. For each byte of an input stream the Sarwate algorithm performs the following: (i) an XOR operation between a byte read and the most significant byte of the current CRC value; (ii) a table lookup; (iii) a shift operation on the current CRC value; and (iv) an XOR operation between the

shifted CRC value and the word read from the table. In contrast, for every byte of an input stream the Slicing-by-8 algorithm performs only a table lookup and an XOR operation. This is the reason why the Slicing-by-8 algorithm is faster than the Sarwate algorithm. Detailed description and proof of correctness of Slicing-by-8 can be found in reference [16].

3.3 Interleaving data copies with the CRC generation process

Following CRC, the next big overhead in iSCSI processing is data copy. Data copy is a memory access-intensive operation, and as such its performance depends on the memory subsystem used. In this respect, data copy differs from the CRC generation process since the latter is compute-intensive and scales with the CPU clock. To further speed up data touching operations beyond CRC as discussed in Sections 3.1 and 3.2, we investigate how data copies and CRC generation take place in iSCSI.

On the inbound path, a packet which is part of an iSCSI PDU is copied from the network buffer to its appropriate offset in the SCSI buffer. In a typical software Linux iSCSI implementation, the placement of the data is controlled by the iSCSI layer, and is performed based on the information contained in the iSCSI PDU header. The copy operation by itself, however, is performed by the sockets/TCP layer. Once the entire PDU payload is placed in the SCSI buffer, the iSCSI layer computes and validates a CRC value over the entire payload. On the outbound path, the iSCSI layer calculates CRC over the entire PDU payload. It then uses the kernel sockets layer to send out this PDU as multiple TCP packets. Again, the CRC is implemented at the iSCSI layer, while the copy is a part of the kernel sockets layer.

Thus, if iSCSI is implemented in a strictly layered fashion over a kernel sockets interface to TCP/IP, copy and CRC are treated as two separate operations. However, if copy and CRC are interleaved, the combined operation results in better system performance because of two reasons:

- Parallelism among the CRC generation (compute-intensive) and data copy (memory access-intensive) operations.
- Warming of the cache memory since the data upon which CRC is computed is transferred to the cache as the copy operation runs ahead.

In essence, we apply the principle of Integrated Layer Processing (ILP) [6] in order to interleave the iSCSI CRC generation process with the data copy operations. In our approach, the sockets/TCP layer computes a CRC over the payload while copying the data between the sockets/TCP buffers and the SCSI buffers. On the

inbound path, the generated CRC value is pushed up to the iSCSI layer for validation. On the outbound path, it is inserted into the iSCSI stream. Figure 10 illustrates how the sequential and interleaved copy-CRC operations take place over time.

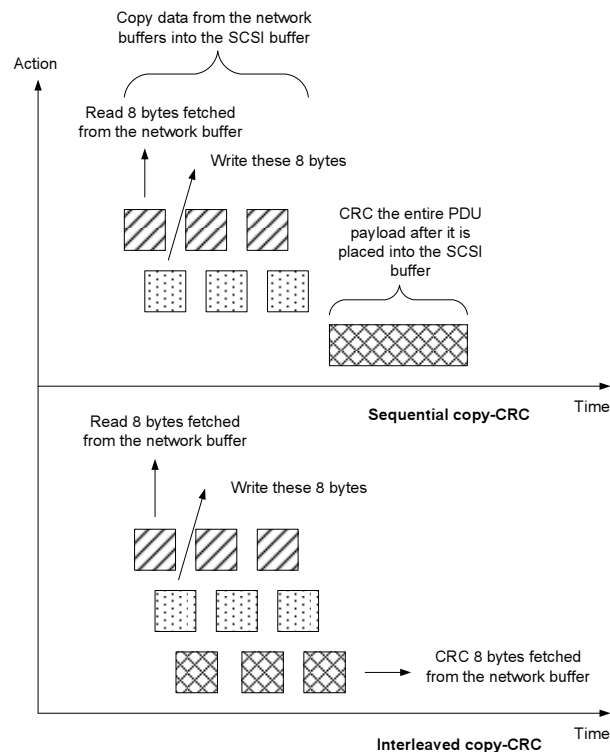


Figure 10: Sequential versus interleaved data copies and CRC

To enable our interleaved copy-CRC optimization, the interface between the iSCSI and TCP layers needs to be modified. This interface needs to allow the calculation of a CRC value on some specified set of bytes while performing a copy. If iSCSI markers are disabled [30], then the bytes that are copied are the same as the bytes on which the CRC is computed. However, when markers are enabled, the copy and CRC bytes are different, since iSCSI CRC does not cover markers. Solving the marker problem is fairly simple. As explained in Section 2, the iSCSI layer creates a scatter-gather list that describes the SCSI buffer elements where the PDU payload is to be copied from or copied to. If the payload contains markers, then the scatter-gather list also includes descriptors pointing to the scratch buffers that receive/source marker bytes. Each such list element can use a single bit 'C' in its descriptor to differentiate between marker versus non-marker bytes. If the bit C is equal to 1, this implies that the socket/TCP layer must validate the data pointed to

by the descriptor. If C is equal to 0, this implies that only a copy operation should be performed on the data without CRC validation. With this scheme, skipping markers for CRC computation becomes easy. The iSCSI layer can simply set C equal to 0 for list elements that describe marker bytes and 1 for all other bytes.

Interleaving data copies and the CRC generation process does not typically modify the functional behavior of the iSCSI protocol stack. This is the case for most stack implementations, which use intermediate buffers for validating PDUs before copying these PDUs into their final buffers (e.g., application or file cache buffers). In our approach, the data copies into these intermediate buffers are interleaved with the CRC generation process. Some stack implementations, however, avoid the extra copy by directly copying the PDU to their application/file cache buffers from the socket/TCP buffers. If the PDU data is corrupted, then these stacks can end up polluting the application/file cache buffers. This can happen if the data is not validated before the copy. Interleaving copy and CRC cannot be used in such stacks if such pollution cannot be tolerated. However, for implementations where the application/file system does not make any assumptions about the buffer contents, the interleaving optimization can still be applied.

4. The iSCSI fast-path

4.1 User-level Sandbox Environment

There are several open-source prototype implementations of iSCSI (e.g. [22], [32]). Some of the earlier work in this area [14, 15, 27, 28] has used these implementations to analyze the iSCSI performance characteristics. These implementations, however, operate on top of standard, unmodified TCP/IP stacks. To evaluate iSCSI in an environment where TCP/IP optimizations like header splitting and interrupt coalescing [4, 26] are present, we took the alternative approach of implementing our own iSCSI fast-path in a user-level sandbox environment. This sandbox environment is closely coupled with an optimized TCP/IP implementation developed at Intel labs [26]. Our implementation is compliant with iSCSI RFC 3720 [30], and includes all protocol level checks in the fast-path as indicated by the specification of the protocol.

There are several benefits associated with a user-level sandbox implementation of a protocol. First, the time required for implementing and testing new ideas in a sandbox is typically much smaller than the time required for a kernel-level prototype. Second, it is easier to run the sandbox implementation on different processor and platform simulators in order to study the scaling of processing costs with architectural

improvements on processors. The main drawback of a sandbox implementation is that it cannot put bits on the wire and thus cannot interact with real world protocol stacks. However, we believe that such an implementation is a useful first step in analyzing complex protocols, especially when protocols are implemented in the kernel.

4.2 Implementation

Our sandbox implementation includes the iSCSI fast-path code for read and write commands, packet data structures, an interface to the SCSI layer, and an interface to the sockets/TCP layer. Our implementation is optimized to align key data structures along cache lines and uses pre-fetching of data structures wherever possible to avoid high memory access latencies. Similarly, SCSI buffers are page aligned and initialized so that the operating system can page-in the buffers. This emulates the effect of pre-pinned SCSI buffers in real implementations so that there are no page faults during the fast path. Before running the code of interest, the test application purges the fast-path data structures out of the cache memory. It then pre-fetches the packet headers or other data structures in order to warm the cache. For example, for inbound read processing operations, the application warms packet headers in order to emulate the effect of TCP header processing. Similarly, SCSI buffers are purged out of the cache or warmed based on whether the run is emulating cold data or recently created application warm data.

For studying the processing costs associated with the incoming (data-in) PDUs of read commands, our initialization code creates state at each layer (i.e., the SCSI, iSCSI and sockets/TCP layer) to emulate the outstanding read commands sent to a target. This includes creating SCSI command structures, the SCSI buffer, iSCSI session information, and command contexts. Incoming TCP segments that make up an iSCSI data PDU and a status PDU are created at the TCP/sockets layer. State is created at the TCP layer as if TCP processing is over and the TCP payload is queued into socket buffers. The cache memory is then purged and, if required, any warming of the cache is done as described earlier. The test application is now ready to execute and measure the fast path, as described in Section 2.1.

For studying the processing costs associated with the outgoing (data-out) PDUs of write commands, our initialization code creates state that emulates outstanding SCSI write commands. This includes creating the SCSI commands and inserting them into the iSCSI session queues based on a Logical Unit Number (LUN). The fast-path then measures the cost of sending out unsolicited data-out PDUs for the write commands. For solicited data-out PDUs, the

initialization code also creates state as if R2T PDUs were received from the target soliciting specific portions of the SCSI write data. The fast path then measures the cost to send out solicited data-out PDUs to the target, as described in Section 2.2.

4.3 Measurement and Simulation techniques

We measure the processing cost of executing the iSCSI stack using two techniques. The first technique runs the stack on a real machine. We use the RDTSC and CPUID instructions [11] of the IA32 processor architecture to measure the cycles spent in the fast path code. To minimize operating system interruptions, our implementation ran at real-time priority and suspended itself before each performance run in order to keep the system stable. Using processor performance counters, we were also able to find out other interesting statistics like instructions retired per PDU, number of second level (L2) cache misses per PDU, and average cycles per instruction (CPI).

The second technique we used involves examining our iSCSI implementation on an instruction-by-instruction basis by running it on a cycle-accurate CPU simulator. The simulator allows us to take a closer look at the micro-architectural behavior of the protocol on a particular processor family. This helps us determine portions of the code that result in cache misses and optimize the code by issuing pre-fetches wherever possible. Simulator runs also help in projecting protocol performance on future processors and platforms. In this way we can determine how different optimizations scale with architectural or clock-speed improvements.

5. Evaluation

5.1 Analysis of iSCSI processing

We begin our evaluation by examining how existing iSCSI implementations perform (i.e., implementations with Sarvate CRC and no copy-CRC interleaving). As mentioned earlier iSCSI processing involves four main components – data structure manipulation, CRC generation, data copies, and marker processing. In this section we also refer to data structure manipulation as ‘protocol processing’ (even though CRC generation and copies are also part of the protocol processing). Our implementation supports all features except markers. Since we started our investigation on a pure software implementation that uses standard kernel sockets for interfacing iSCSI with TCP, we did not implement markers since marker benefits are tied to the close coupling between iSCSI and TCP.

For both the direct execution (i.e., the execution on a real machine) and the CPU simulation runs, we used a 1.7 GHz Intel® Pentium® M processor, with a 400 MHz FSB, and a single channel DDR-266 memory subsystem. The workload consisted of a single session with 8 connections, and 40 commands (read and write commands) issued on a round-robin fashion over the connections. Table 2 shows our parameters for read command processing.

Parameter	Value
Maximum receive data segment length	8KB
Max burst length	256KB
Data PDU in order	No
Data sequence in order	No
Header digest	CRC32c
Data digest	CRC32c

Table 2: Read command parameters

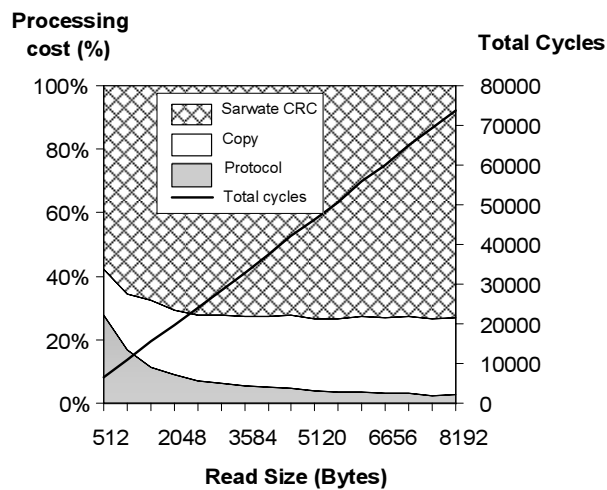


Figure 11: Data-in PDU processing cost (with Sarwate CRC)

Figure 11 shows the iSCSI processing cost for a single data-in (read) PDU at the initiator. The horizontal axis represents the size of the read I/O command issued by the initiator. The primary vertical axis (i.e., the axis on the left) represents the relative processing costs of the protocol, the CRC generation process, and the data copies. The secondary vertical axis (i.e., the axis on the right) represents the absolute number of cycles spent on iSCSI processing for a single data-in PDU. Since the read I/O workload size varies from 512B to 8KB, and the PDU size is set to 8KB, this implies that each iSCSI read command results in the reception of a single data-in PDU. In this experiment we also enabled the ‘phase-collapse’ feature of iSCSI. Thus each data-in PDU also carried a status response from the target.

For the smallest I/O size (i.e., 512B), the protocol cost is about 28% of the total cost, the copy cost is about 14% of the total cost, whereas CRC accounts for the remaining 58%. As the PDU size increases, the protocol cost remains the same on a per PDU basis, whereas the CRC and copy costs increase. For the largest I/O size (i.e., 8KB) the protocol cost is barely 3% of the total cost, whereas CRC is 73% of the total cost and copy accounts for 24%.

The protocol cost is paid once per PDU. It is about 1800 cycles and remains constant for each command with a slight bump at the page boundary (i.e., at 4KB), since crossing a page boundary involves adding another scatter element to the SCSI buffer. On the other hand, the CRC cost is paid on a per byte basis and increases linearly as the I/O size increases. For a workload of 8KB, the total CRC cost is about 54000 cycles, or about 6.5 cycles per byte, while the copy cost is about 2.15 cycles per byte. Thus for the 8KB PDU size which is a common PDU size, the CRC cost completely dominates the iSCSI protocol processing cost, and is significantly more than the copy cost.

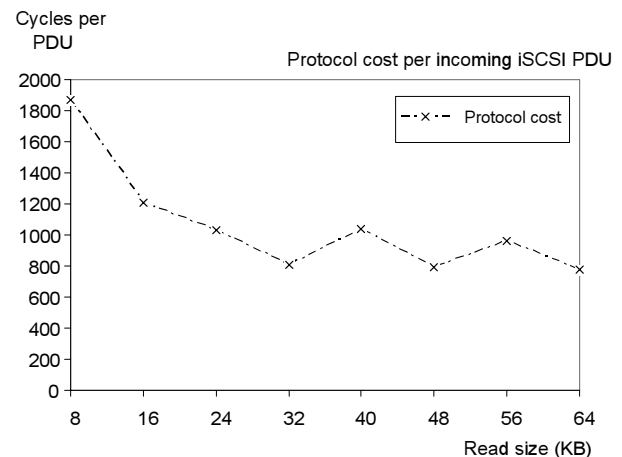


Figure 12: Data-in PDU Protocol processing cost

To get a deeper understanding of the cycles spent on protocol processing and the CRC generation process, we ran the same experiment on a cycle accurate simulator and measured the number of instructions executed on a per PDU basis and the number of L2 misses occurring per PDU. The protocol execution path for an 8KB PDU was about 438 instructions long, with about 6 L2 misses per PDU. These L2 misses can be attributed to accessing an iSCSI connection context, a command context, a score-boarding data structure for out-of-order PDUs, a SCSI buffer descriptor, and a SCSI context to store the response received from the target. The instruction path length for CRC is about 8 instructions per byte. We measured that it takes about 6.55 cycles per byte to compute a CRC value, with a

‘Cycles per Instruction’ (CPI) value of slightly less than 1. Next, we ran our simulator with a ‘perfect cache’ option enabled in order to simulate the ideal case in which all contexts and data are warm in the cache. In this way we were able to measure the best-case protocol processing cost that needs to be paid for processing a single data-in PDU with an embedded response. This cost was about 430 cycles which agrees with the instruction path length of 438.

In order to determine how close the protocol processing cost for a data-in PDU can approach the asymptote of 430 cycles, we ran the same experiment but with the read I/O workload size ranging from 8KB to 64 KB. Since the PDU size is fixed at 8KB, this means that each iSCSI command resulted in the reception of 1 to 8 data-in PDUs. For these experiments the last data-in PDU contained the embedded status.

Figure 12 shows the average protocol processing cost for an 8KB PDU. As seen in the graph, this cost drops down from about 1800 cycles to about 770 cycles. This drop can be attributed to the fact that some of the contexts like the SCSI descriptors are now warm in the cache. Thus, subsequent PDUs after the first PDU benefit from cache warming. The larger the read size is, the lower the per-PDU protocol cost becomes. On the other hand, the per byte cost of CRC remains the same whether the initiator reads 64 KB of data as a single 64KB read operation or 8 8KB read operations. Thus, while data structure manipulation becomes more efficient with larger read sizes, the CRC cost remains the same.

The protocol processing cost of write commands is smaller than the cost of read commands, while the CRC processing cost is the same since CRC generation incurs the same per byte cost independent of the direction of the data. In addition most stacks completely avoid or have at most one copy operation on the outbound path. Because of these reasons, CRC is a bigger bottleneck for write commands as compared to read commands.

5.2. Impact of Optimizations

To evaluate the performance benefits of the software optimizations we discussed in Section 3, we added the Slicing-by-8 (SB8) CRC implementation into our sandbox iSCSI stack. The test system and workload for these experiments were the same as described in Section 5.1. Figure 14 compares the performance of iSCSI read runs with two different CRC generation algorithms. The read I/O workload size ranges from 512B to 8KB. The horizontal axis represents the read I/O size, while the vertical axis shows the total cycles spent on computing a CRC over that I/O size.

As seen in the graph, the Slicing-by-8 algorithm requires lesser cycles than the Sarwate algorithm to compute the CRC at all data points. Specifically, for the 8KB data point, the Slicing-by-8 algorithm takes about 2.15 cycles per byte to generate a CRC value, while the Sarwate algorithm takes 6.55 cycles per byte. Thus, Slicing-by-8 accelerates the CRC generation process by a factor of 3, as compared to Sarwate. The Sarwate algorithm requires executing 35 IA32 instructions in order to validate 32 bits of data whereas the Slicing-by-8 algorithm requires 13 instructions only. This is the reason why the Slicing-by-8 algorithm is three times faster than the Sarwate algorithm.

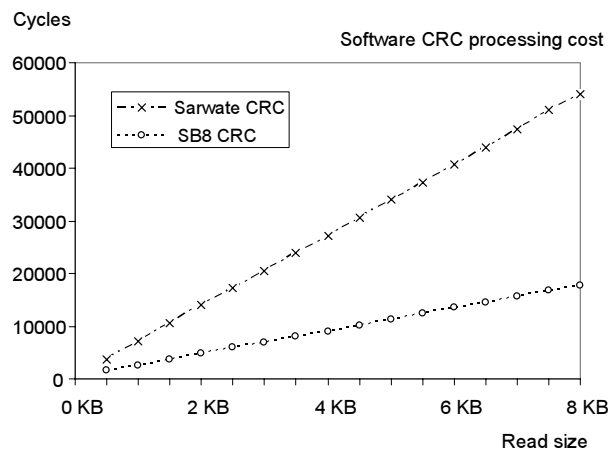


Figure 14: SB8 CRC versus Sarwate CRC

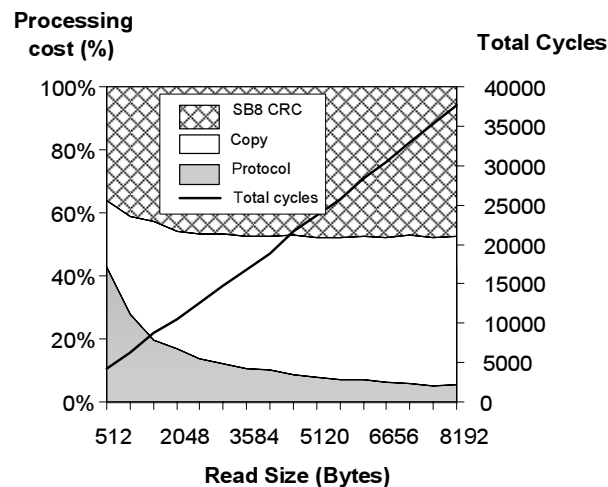


Figure 15: Data-in PDU processing cost (with SB8 CRC)

Figure 15 shows the performance profile characterizing the processing of a data-in PDU where the CRC generation algorithm is Slicing-by-8. Operating at 2.15 cycles per byte, the Slicing-by-8 algorithm demonstrates the same cost as the data

copies. For the 8KB PDU size, each of data copies and CRC generation represent about 47% of the total processing cost, while the protocol processing is about 5%. Comparing these numbers with the costs of Figure 11 (similar profile but with the Sarwate CRC algorithm), we can see that the total processing cost for an 8KB PDU has now decreased from 73761 cycles to about 37579 cycles, resulting in two times faster iSCSI processing.

We also performed a second group of experiments to evaluate the impact of interleaving data copies with the CRC generation process. We modified our sandbox implementation in order to support a new iSCSI/TCP interface and the interleaved copy-CRC operations as described in Section 3.3.

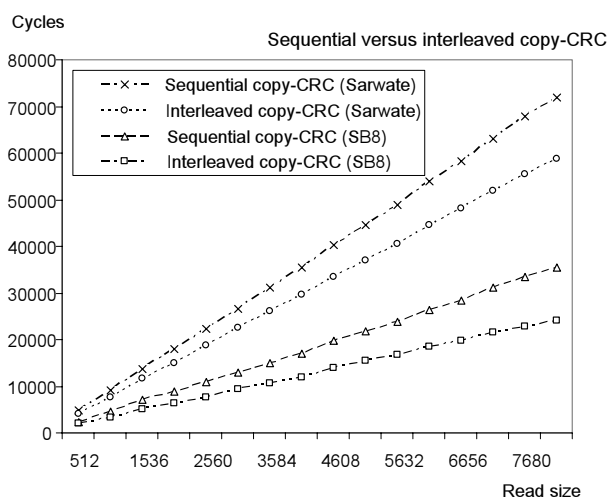


Figure 16: Interleaved Copy-CRC cost

Figure 16 shows the measured cost of sequential copy and CRC operations for both the Sarwate and Slicing-by-8 algorithms while executing iSCSI reads. It also shows the interleaved copy-CRC cost. As seen in the figure, for both CRC algorithms, interleaving copy with CRC reduces the overall cycle cost for the two operations. For the interleaved Sarwate CRC and copy, the total cycle reduction is about 13000 cycles or 18%. For the interleaved Slicing-by-8 CRC and copy, the cycle reduction is about 11570 cycles corresponding to a performance improvement of 32%.

We then extended our measurements and analysis to an entire storage stack consisting of our iSCSI implementation, and an optimized TCP/IP stack implementation. Our goal was to understand the impact of the optimizations (i.e., the Slicing-by-8 algorithm and the interleaved copy-CRC) on the performance of the entire stack. Figure 17 shows the overall throughput as seen at the iSCSI layer for different values of a read I/O workload size. The x-axis represents the size of the read command, while the y-axis shows the achieved

iSCSI throughput in MB/s across the range of read sizes. The topmost line depicts the maximum iSCSI line rate that can be achieved for a 10Gbps Ethernet link. As seen in the figure, the iSCSI throughput improves from 175 MB per second (or 1.4 Gbps) to about 445 MB per second (or 3.6 Gbps) when both the optimizations are turned on.

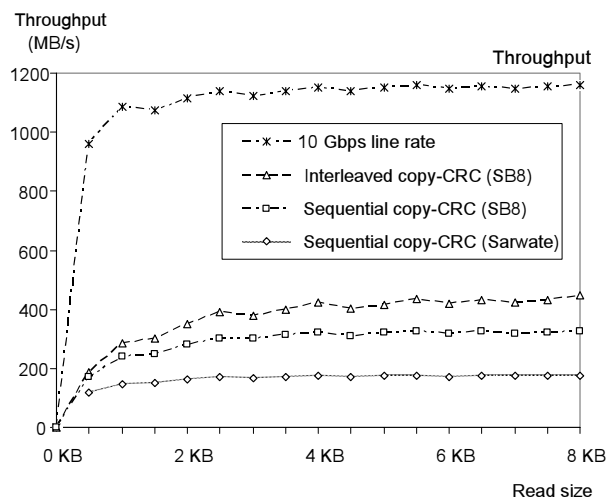


Figure 17: Projected software Read throughput

To further evaluate our Slicing-by-8 algorithm, we measured the impact of Slicing-by-8 on a real-world Linux iSCSI stack developed by UNH [32]. We modified this implementation by replacing the Sarwate algorithm with the Slicing-by-8 algorithm. Our experimental environment consisted of a 3 GHz single-threaded uni-processor Xeon server board with a 533 MHz FSB and a 64bit/133 MHz PCI bus for the initiator. The initiator was running Linux kernel 2.4.20. The iSCSI initiator stack was connected to 3 UNH iSCSI targets through gigabit NICs. We measured the normalized throughput (bits/Hz) of iSCSI with the two CRC algorithms and found that for an 8KB I/O workload, replacing the Sarwate algorithm with Slicing-by-8 results in an increase in the normalized throughput by 15%.

The reason why Slicing-by-8 does not result in a similar improvement like the one demonstrated in the sandbox environment is because the TCP/IP and SCSI overheads are significant in the Linux 2.4 kernel [15]. The 2.6 kernel, though, supports a more optimized implementation of the TCP/IP and SCSI protocols. Unfortunately at the time of writing we did not have access to a modified 2.6 Linux kernel with TCP/IP optimizations. We believe that, as TCP/IP stack implementations become more optimized in the future the CRC overhead in iSCSI will stand out, and the impact of Slicing-by-8 will be greater. As future work, we plan to test our optimizations using the 2.6 kernel.

6. Related Work

Several studies on the performance of iSCSI have been published [1, 14, 15, 17, 21, 24, 27, 28]. These studies have focused on comparing software with hardware implementations in LAN and WAN environments. Sarkar et al [27] compared the performance of three competing approaches to implementing iSCSI. Aiken et al [1] evaluated a commercial implementation of iSCSI and found it to be competitive with a Fiber Channel interface at gigabit speeds. Khosravi et al [15] studied the architectural characteristics of iSCSI processing including CPI, cache effects, and instruction path length. Our work builds on all the previous work described above. We implemented our own iSCSI fast path and demonstrated a performance improvement on the iSCSI processing throughput using two new optimizations.

Radkov et al [24], and Lu et al [17] have compared iSCSI and NFS/SMB-based storage. Magoutis et al [19] performed a thorough comparison and analysis of the DAFS and NFS protocols. Our work is narrower in scope in that it looks only at iSCSI-based storage, but takes a much deeper dive into the performance issues of iSCSI.

Efficient implementation of the CRC generation process has been the subject of substantial amount of research [2, 3, 7-10, 12, 20, 25, 29, 31, 34]. Software-based CRC generation has been investigated in [8-10, 12, 25, 29, 34]. Among these algorithms the most commonly used today is the one proposed by Sarwate [29]. Feldmeier [8] motivated by the fact that table-driven solutions are subject to cache pollution presented an alternative software technique that avoids the use of lookup tables. Our algorithm is distinguished from [8, 9, 25, 29, 34] by the fact it can ideally read arbitrarily large amounts of data at a time.

The concept of parallel table lookups which we use in our algorithm also appears in early CRC5 implementations [10] and in the work done by Braun and Waldvogel [3] on performing incremental CRC updates for IP over ATM networks. Our work is distinguished from [3, 10] in that our algorithms reuse the same lookup tables in each iteration, thus keeping the memory requirement of CRC generation at reasonable level. On the other hand, if the contribution of each slice to the final CRC value is computed using the square and multiply technique as in the work by Doering and Waldvogel [7], the processing cost may be too high in software.

Our algorithm also bears some resemblance with a recent scheme published by Joshi, Dubey and Kaplan [12]. Like our algorithm the Joshi-Dubey-Kaplan scheme calculates the remainders from multiple slices

of a stream in parallel. The Joshi-Dubey-Kaplan scheme has been designed to take advantage of the 128-bit instruction set extensions to IBM's PowerPC architecture. In our contrast our algorithm does not make any assumptions about the instruction set used.

7. Conclusions and Future Work

In this paper, we report on an in-depth analysis of the performance of IP-based networked block storage systems based on an implementation of the iSCSI protocol. Our data shows that CRC is by far the biggest bottleneck in iSCSI processing, and its impact will increase even further as TCP/IP stacks become more optimized in the future. We demonstrate significant performance improvement through a new software CRC algorithm that is 3 times faster than current industry standard algorithm and show that enhancements to the iSCSI/TCP interface can result in significant performance gains.

We expect that in the future, iSCSI performance will demonstrate near linear scaling with the number of CPU cores available in a system and will support data rates greater than what a single 10 Gigabit Ethernet interface will provide. Three factors lead us to this conclusion: (i) the increasing commercial availability of dual-core and multiple-core CPUs; (ii) evolving operating system technologies such as receive-side scaling that allow distribution of network processing across multiple CPUs; and (iii) the use of multiple iSCSI connections between an initiator and one or more storage targets as supported in many iSCSI implementations today. In the future, we would like to extend our analysis to study the performance and scalability of iSCSI across multiple CPU cores, as well as the application-level performance of iSCSI storage stacks for both transaction-oriented applications as well as backup and recovery applications.

References

- [1] S. Aiken, D. Grunwald, and A. Pleszkun, "A performance analysis of the iSCSI protocol", *Proceedings of the Twentieth IEEE/NASA International Conference on Mass Storage Systems and Technologies (MSST 2003)*, San Diego, CA, April, 2003.
- [2] G. Albertengo, and R. Sisto, "Parallel CRC Generation", *IEEE Micro*, Vol. 10, No. 5., pg. 63-71, 1990.
- [3] F. Braun, and M. Waldvogel, "Fast Incremental CRC Updates for IP over ATM Networks", *Proceedings of the IEEE Workshop on High Performance Switching and Routing (HPSR 2001)*, Dallas, TX, May, 2001.
- [4] J. Chase, A. Gallatin, and K. Yocum, "End-System Optimizations for High-Speed TCP", *IEEE Communications Magazine*, Vol. 39, No.4, pg. 68-74, 2001.

- [5] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead", *IEEE Communications Magazine*, Vol. 27, No.6, pg. 23-29, 1989.
- [6] D. Clark, and D. Tennenhouse, "Architectural Considerations for a new generation of protocols", *Proceedings of the ACM symposium on Communications Architectures and Protocols*, Pennsylvania, PA, September 1990.
- [7] A. Doering, and M. Waldvogel, "Fast and flexible CRC calculation", *Electronics Letters*, Vol. 40, No.1, pg. 10-11, 2004.
- [8] D. Feldmeier, "Fast Software Implementation of Error Correcting Codes", *IEEE/ACM Transactions on Networking*, Vol. 6, No.3, pg. 640-651, 1995.
- [9] G. Griffiths, and G. C. Stones, "The Tea-leaf Reader Algorithm: An Efficient Implementation of CRC-16 and CRC-32", *Communications of the ACM*, Vol. 30, No. 7, pg. 617-620, July 1987.
- [10] C. M. Heard, "AAL2 CPS-PH HEC calculations using table lookups", *Public Domain Source Code*, available at ftp://ftp.vvnet.com/aal2_hec/crc5.c
- [11] "IA-32 Intel Architecture Software Developer's Manual, Volumes 2A and 2B: Instruction Set Reference", <ftp://download.intel.com/design/Pentium4/manuals/>
- [12] S. M. Joshi, P. K. Dubey, and M. A. Kaplan, "A New Parallel Algorithm for CRC Generation", *Proceedings of the International Conference on Communications*, New Orleans, LA, June, 2000.
- [13] H. Keng and J. Chu, "Zero-copy TCP in Solaris", *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January, 1996
- [14] H. Khosravi, and A. Foong, "Performance Analysis of iSCSI And Effect of CRC Computation", *Proceedings of the First Workshop on Building Block Engine Architectures for Computers and Networks (BEACON 2004)*, Boston, MA, October, 2004.
- [15] H. Khosravi, A. Joglekar, and R. Iyer, "Performance Characterization of iSCSI Processing in a Server Platform", *Proceedings of the Twenty Fourth IEEE International Performance Computing and Communications Conference (IPCCC 2005)*, Phoenix, AZ, April 2005.
- [16] M. E. Kounavis and F. Berry, "A Systematic Approach to Building High Performance, Software-based, CRC Generators", *Proceedings of the Tenth IEEE International Symposium on Computers and Communications (ISCC 2005)*, Cartagena, Spain, June, 2005.
- [17] Y. Lu, and D. Du, "Performance Study of iSCSI-Based Storage Subsystems", *IEEE Communications Magazine*, Vol. 41, No. 8, pg. 76-82, 2003.
- [18] K. Magoutis, S. Addetia, A. Fedorova, and M. Seltzer, "Making the Most out of Direct-Access Network Attached Storage", *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST 2003)*, San Francisco, CA, 2003.
- [19] K. Meth, and J. Satran, "Features of the iSCSI Protocol", *IEEE Communications Magazine*, Vol. 41, No. 8, pg. 72-75, 2003.
- [20] M. C. Nielson, "Method for High Speed CRC computation", *IBM Technical Disclosure Bulletin*, Vol. 27, No. 6, pg. 3572-3576, 1984.
- [21] W. T. Ng, H. Sun, B. Hillyer, E. Shriver, E. Gabber, and B. Ozden, "Obtaining High Performance for Storage Outsourcing", *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, Monterey, CA, 2002.
- [22] "Open iSCSI project", *Public Domain Source Code*, available at <http://www.open-iscsi.org/>
- [23] A. Perez, "Byte-wise CRC Calculations", *IEEE Micro*, Vol. 3, No. 3, pg. 40-50, 1983.
- [24] P. Radkov, L. Yin, P. Goyal, P. Sarkar, and P. Shenoy, "A Performance Comparison of NFS and iSCSI for IP-Networked Storage", *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, San Francisco, CA, March 2004.
- [25] T. V. Ramabadran, and S. V. Gaitonde, "A Tutorial on CRC Computations", *IEEE Micro*, Vol. 8, No. 4, pg. 62-75, 1988.
- [26] G. Regnier, S. Makineni, R. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, and D. Newell, "TCP Onloading for Datacenter Server: Perspectives and Challenges", *IEEE Computer Magazine*, Vol. 37, No. 11, pg. 48-58, November 2004.
- [27] P. Sarkar, S. Uttamchandani, K. Voruganti, "Storage over IP: When Does Hardware Support Help?", *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST 2003)*, San Francisco, CA, March 2003.
- [28] P. Sarkar, and K. Voruganti, "IP Storage: The Challenge Ahead", *Proceedings of the Nineteenth IEEE/NASA International Conference on Mass Storage Systems and Technologies (MSST 2002)*, College Park, MD, April, 2002.
- [29] D. V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Lookup", *Communications of the ACM*, Vol. 31, No 8, pg.1008-1013, 1988.
- [30] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)", *RFC 3720*, April 2004.
- [31] M. D. Shieh, M. H. Sheu, C. H. Chen and H. F. Lo, "A systematic Approach for Parallel CRC Computations", *Journal of Information Science and Engineering*, Vol. 17, pg. 445-461, 2001
- [32] UNH iSCSI project, *Public Domain Source Code*, available <http://unh-iscsi.sourceforge.net/>
- [33] K. Voruganti, and P. Sarkar, "An Analysis of Three Gigabit Storage Networking Protocols", *Proceedings of the Twentieth IEEE International Performance Computing and Communications Conference (IPCCC 2001)*, Phoenix, AZ, April 2001.
- [34] R. Williams, "A painless guide to CRC Error Detection Algorithms", *Technical Report*, available at: ftp://ftp.rocksoft.com/papers/crc_v3.txt, 1993.

Intel and Pentium are trademarks or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries. Copyright © 2005, Intel Corporation.

TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization

Navendu Jain[†], Mike Dahlin[†], and Renu Tewari[§]

[†]*Department of Computer Sciences, University of Texas at Austin, Austin, TX, 78712*

[§]*IBM Almaden Research Center, 650 Harry Road, San Jose, CA, 95111*

{nav,dahlin}@cs.utexas.edu, tewarir@us.ibm.com

Abstract

We present TAPER, a scalable data replication protocol that *synchronizes* a large collection of data across multiple geographically distributed replica locations. TAPER can be applied to a broad range of systems, such as software distribution mirrors, content distribution networks, backup and recovery, and federated file systems. TAPER is designed to be bandwidth efficient, scalable and content-based, and it does not require prior knowledge of the replica state. To achieve these properties, TAPER provides: i) four pluggable redundancy elimination phases that balance the trade-off between bandwidth savings and computation overheads, ii) a *hierarchical hash tree* based directory pruning phase that quickly matches identical data from the granularity of directory trees to individual files, iii) a content-based similarity detection technique using *Bloom filters* to identify similar files, and iv) a combination of coarse-grained chunk matching with finer-grained block matches to achieve bandwidth efficiency. Through extensive experiments on various datasets, we observe that in comparison with rsync, a widely-used directory synchronization tool, TAPER reduces bandwidth by 15% to 71%, performs faster matching, and scales to a larger number of replicas.

1 Introduction

In this paper we describe TAPER, a redundancy elimination protocol for replica synchronization. Our motivation for TAPER arose from building a federated file system using NFSv4 servers, each sharing a common system-wide namespace [25]. In this system, data is replicated from a master server to a collection of servers, updated at the master, periodically synchronized to the other servers, and read from any server via NFSv4 clients. Synchronization in this environment requires a protocol that minimizes both network bandwidth consumption and end-host overheads. Numerous applications have similar requirements: they require replicating and synchronizing a large collection of data across multiple sites, possibly over low-bandwidth links. For example, software distribution mirror sites, synchronizing personal

systems with a remote server, backup and restore systems, versioning systems, content distribution networks (CDN), and federated file systems all rely on synchronizing the current data at the source with older versions of the data at remote target sites and could make use of TAPER.

Unfortunately, existing approaches do not suit such environments. On one hand, protocols such as delta compression (e.g., vcdiff [14]) and snapshot differencing (e.g., WAFL [11]) can efficiently update one site from another, but they require *a priori* knowledge of which versions are stored at each site and what changes occurred between the versions. But, our environment requires a *universal* data synchronization protocol that interoperates with multi-vendor NFS implementations on different operating systems without any knowledge of their internal state to determine the version of the data at the replica. On the other hand, hash-based differential compression protocols such as rsync [2] and LBFS [20] do not require *a priori* knowledge of replica state, but they are inefficient. For example, rsync relies on path names to identify similar files and therefore transfers large amounts of data when a file or directory is renamed or copied, and LBFS's single-granularity chunking compromises efficiency (a) by transferring extra metadata when redundancy spanning multiple chunks exists and (b) by missing similarity on granularities smaller than the chunk size.

The TAPER design focuses on providing four key properties in order to provide speed, scalability, bandwidth efficiency, and computational efficiency:

- **P1:** Low, re-usable computation at the source
- **P2:** Fast matching at the target
- **P3:** Find maximal common data between the source and the target
- **P4:** Minimize total metadata exchanged

P1 is necessary for a scalable solution that can simultaneously synchronize multiple targets with a source. Similarly, P2 is necessary to reduce the matching time and, therefore, the total response time for synchronization. To support P2, the matching at the target should be based on

indexing to identify the matching components in $O(1)$ time. The last two, P3 and P4, are both indicators of bandwidth efficiency as they determine the total amount of data and the total metadata information (hashes etc.) that are transferred. Balancing P3 and P4 is the key requirement in order to minimize the metadata overhead for the data transfer savings. Observe that in realizing P3, the source and target should find common data across all files and not just compare file pairs based on name.

To provide all of these properties, TAPER is a multi-phase, hierarchical protocol. Each phase operates over decreasing data granularity, starting with directories and files, then large chunks, then smaller blocks, and finally bytes. The phases of TAPER balance the bandwidth efficiency of smaller-size matching with the reduced computational overhead of lesser unmatched data. The first phase of TAPER eliminates all common files and quickly prunes directories using a content-based *hierarchical hash tree* data structure. The next phase eliminates all common content-defined chunks (CDC) across all files. The third phase operates on blocks within the remaining unmatched chunks by applying a similarity detection technique based on Bloom filters. Finally, the matched and unmatched blocks remaining at the source are further delta encoded to eliminate common bytes.

Our main contributions in this paper are: i) design of a new hierarchical hash tree data structure for fast pruning of directory trees, ii) design and analysis of a similarity detection technique using CDC and Bloom filters that compactly represent the content of a file, iii) design of a combined CDC and *sliding block* technique for both coarse-grained and fine-grained matching, iv) integrating and implementing all the above techniques in TAPER, a multi-phase, multi-grain protocol, that is engineered as pluggable units. The phases of TAPER are pluggable in that each phase uses a different mechanism corresponding to data granularity, and a phase can be dropped all together to trade bandwidth savings for computation costs. And, v) a complete prototype implementation and performance evaluation of our system. Through extensive experiments on various datasets, we observe that TAPER reduces bandwidth by 15% to 71% over rsync for different workloads.

The rest of the paper is organized as follows. Section 2 provides an overview of the working of sliding block and CDC. These operations form the basis of both the second and third phases that lie at the core of TAPER. The overall TAPER protocol is described in detail in Section 3. Similarity detection using CDC and Bloom filters is described and analyzed in Section 4. Section 5 evaluates and compares TAPER for different workloads. Finally, Section 6 covers related work and we conclude with Section 7.

2 Background

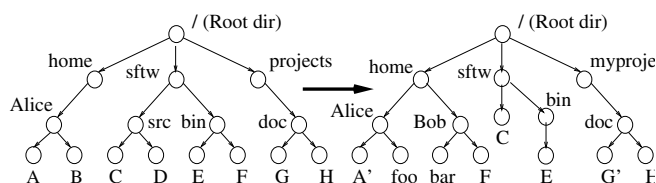


Figure 1: Directory Tree Synchronization Problem: The source tree is shown on the left and the target tree with multiple updates, additions, and renames, is on the right.

In synchronizing a directory tree between a source and target (Figure 1), any approach should efficiently handle all the common update operations on file systems. These include: i) adding, deleting, or modifying files and directories, ii) moving files or directories to other parts of the tree, iii) renaming files and directories, and iv) archiving a large collection of files and directories into a single file (e.g., tar, lib).

Although numerous tools and utilities exist for directory synchronization with no data versioning information, the underlying techniques are either based on matching: i) block hashes or ii) hashes of content-defined chunks. We find that sliding block hashes (Section 2.1) are well suited to relatively fine-grained matching between similar files, and that CDC matching (Section 2.2) is suitable for more coarse-grained, global matching across all files.

2.1 Fixed and Sliding Blocks

In block-based protocols, a fixed-block approach computes the signature (e.g., SHA-1, MD5, or MD4 hash) of a fixed-size block at both the source and target and simply indexes the signatures for a quick match. Fixed-block matching performs poorly because small modifications change all subsequent block boundaries in a file and eliminate any remaining matches. Instead, a sliding-block approach is used in protocols like rsync for a better match. Here, the target, T , divides a file f into non-overlapping, contiguous, fixed-size blocks and sends its signatures, 4-byte MD4 along with a 2-byte rolling checksum (rsync's implementation uses full 16 byte MD4 and 4 byte rolling checksums per-block for large files), to the source S . If an existing file at S , say f' , has the same name as f , each block signature of f is compared with a sliding-block signature of every overlapping fixed-size block in f' . There are several variants of the basic sliding-block approach, which we discuss in Section 6, but all of them compute a separate multi-byte checksum for each byte of data to be transferred. Because this checksum information is large compared to the data being stored, it would be too costly to store all checksums for all offsets of all files in a system, so these

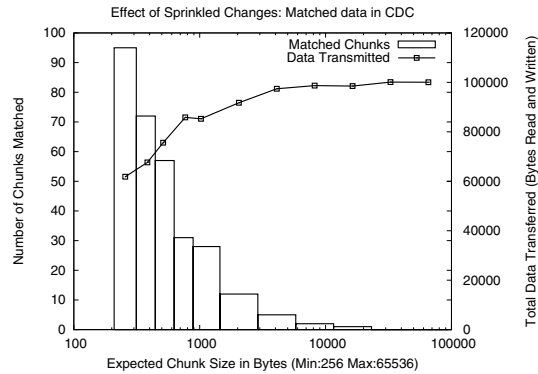


Figure 2: Effect of Sprinkled Changes in CDC. The x-axis is the expected chunk size. The left y-axis, used for the bar graphs, shows the number of matching chunks. The right y-axis, for the line plot, shows the total data transferred

systems must do matching on a finer (e.g., per-file) granularity. As a result, these systems have three fundamental problems. First, matching requires knowing which file f' at the source should be matched with the file f at the target. Rsync simply relies on file names being the same. This approach makes rsync vulnerable to name changes (i.e., a rename or a move of a directory tree will result in no matches, violating property P3). Second, scalability with the number of replicas is limited because the source machine recomputes the sliding block match for every file and for every target machine and cannot re-use any hash computation (property P1). Finally, the matching time is high as there is no indexing support for the hashes: to determine if a block matches takes time of the order of number of bytes in a file as the rolling hash has to be computed over the entire file until a match occurs (property P2). Observe that rsync [2] thus violates properties P1, P2, and P3. Although rsync is a widely used protocol for synchronizing a single client and server, it is not designed for large scale replica synchronization.

To highlight the problem of name-based matching in rsync, consider, for example, the source directory of GNU Emacs-20.7 consisting of 2086 files with total size of 54.67 MB. Suppose we rename only the top level sub-directories in Emacs-20.7 (or move them to another part of the parent tree). Although no data has changed, rsync would have sent the entire 54.67 MB of data with an additional 41.04 KB of hash metadata (using the default block size of 700 bytes), across the network. In contrast, as we describe in Section 3.1.1, TAPER alleviates this problem by performing content-based pruning using a hierarchical hash tree.

2.2 Content-defined Chunks

Content-defined chunking balances the fast-matching of a fixed-block approach with the finer data matching ability of sliding-blocks. CDC has been used in LBFS [20],

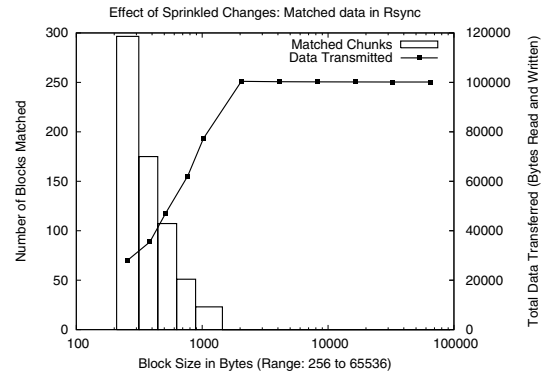


Figure 3: Effect of Sprinkled Changes in Rsync. The x-axis is the fixed block size. The left y-axis, used for the bar graphs, shows the number of matching blocks. The right y-axis, for the line plot, shows the total data transferred.

Venti [21] and other systems that we discuss in Section 6. A chunk is a variable-sized block whose boundaries are determined by its Rabin fingerprint matching a pre-determined marker value [22]. The number of bits in the Rabin fingerprint that are used to match the marker determine the expected chunk size. For example, given a marker 0x78 and an expected chunk size of 2^k , a rolling (overlapping sequence) 48-byte fingerprint is computed. If the lower k bits of the fingerprint equal 0x78, a new chunk boundary is set. Since the chunk boundaries are content-based, a file modification should affect only neighboring chunks and not the entire file. For matching, the SHA-1 hash of the chunk is used. Matching a chunk using CDC is a simple hash table lookup.

Clearly, the expected chunk size is critical to the performance of CDC and depends on the degree of file similarity and the locations of the file modifications. The chunk size is a trade-off between the degree of matching and the size of the metadata (hash values). Larger chunks reduce the size of metadata but also reduce the number of matches. Thus, for any given chunk size, the CDC approach violates properties P3, P4, or both. Furthermore, as minor modifications can affect neighboring chunks, changes sprinkled across a file can result in few matching chunks. The expected chunk size is manually set in LBFS (8 KB default). Similarly, the fixed block size is manually selected in rsync (700 byte default).

To illustrate the effect of small changes randomly distributed in a file, consider, for example, a file (say 'bar') with 100 KB of data that is updated with 100 changes of 10 bytes each (i.e., a 1% change). Figures 2 and 3 show the variations due to sprinkled changes in the matched data for CDC and rsync, respectively. Observe that while rsync finds more matching data than CDC for small block sizes, CDC performs better for large chunk sizes. For a block and expected chunk size of 768 bytes,

rsync matched 51 blocks, transmitting a total of 62 KB, while CDC matched 31 chunks, transmitting a total of 86 KB. For a larger block size of 2 KB, however, rsync found no matches, while CDC matched 12 chunks and transmitted 91 KB. In designing TAPER, we use this observation to apply CDC in the earlier phase with relatively larger chunk sizes.

3 TAPER Algorithm

In this section, we first present the overall architecture of the TAPER protocol and then describe each of the four TAPER phases in detail.

3.1 TAPER Protocol Overview

TAPER is a directory tree synchronization protocol between a source and a target node that aims at minimizing the transmission of any common data that already exists at the target. The TAPER protocol does not assume any knowledge of the state or the version of the data at the target. It, therefore, builds on hash-based techniques for data synchronization.

In general, for any hash-based synchronization protocol, the smaller the matching granularity the better the match and lower the number of bytes transferred. However, fine-grained matching increases the metadata transfer (hash values per block) and the computation overhead. While systems with low bandwidth networks will optimize on the total data transferred, those with slower servers will optimize the computation overhead.

The intuition behind TAPER is to work in phases (Figure 4) where each phase moves from a larger to a finer matching granularity. The protocol works in four phases: starting from a directory tree, moving on to large chunks, then to smaller blocks, and finally to bytes. Each phase in TAPER uses the best matching technique for that size, does the necessary transformations, and determines the set of data over which the matches occur.

Specifically, the first two phases perform coarse grained matching at the level of directory trees and large CDC chunks (4 KB expected chunk size). Since the initial matching is performed at a high granularity, the corresponding hash information constitutes only a small fraction of the total data. The SHA-1 hashes computed in the first two phases can therefore be pre-computed once and stored in a *global* and *persistent* database at the source. The *global* database maximizes matching by allowing any directory, file, or chunk that the source wants to transmit to be matched against any directory, file, or chunk that the target stores. And the *persistent* database enhances computational efficiency by allowing the source to re-use hash computations across multiple targets. Conversely, the last two phases perform matching at the level of smaller blocks (e.g., 700 bytes), so precomputing and storing all hashes of all small blocks

would be expensive. Instead, these phases use *local* matching in which they identify similar files or blocks and compute and *temporarily* store summary metadata about the specific files or blocks currently being examined. A key building block for these phases is efficient *similarity detection*, which we assume as a primitive in this section and discuss in detail in Section 4.

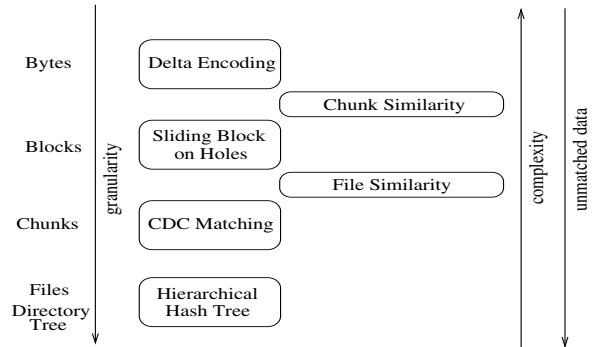


Figure 4: The building blocks of TAPER

3.1.1 Directory Matching

The first phase, directory matching, eliminates identical portions of the directory tree that are common in content and structure (but may have different names) between the source and the target. We define a *hierarchical hash tree* (HHT) for this purpose to quickly find all the exact matching subtrees progressing down from the largest to the smallest directory match and finally matching identical individual files.

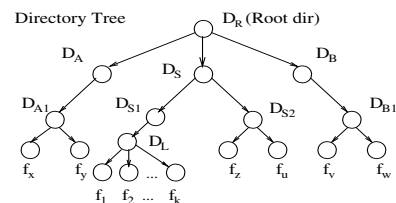


Figure 5: Phase I: Hierarchical Hash Tree

The HHT representation encodes the directory structure and contents of a directory tree as a list of hash values for every node in the tree. The nodes consist of the root of the directory tree, all the internal sub-directories, leaf directories, and finally all the files. The HHT structure is recursively computed as follows. First, the hash value of a file node f_i is obtained using a standard cryptographic hash algorithm (SHA-1) of the contents of the file. Second, for a leaf directory D_L , the hash value $h(D_L)$ is the hash of all the k constituent file hashes, i.e., $h(D_L) = h(h(f_1)h(f_2)...h(f_k))$. Note that the order of concatenating hashes of files within the same directory is based on the hash values and not on the file names. Third, for a non-leaf sub-directory, the hash value captures not only the content as in Merkle trees but also

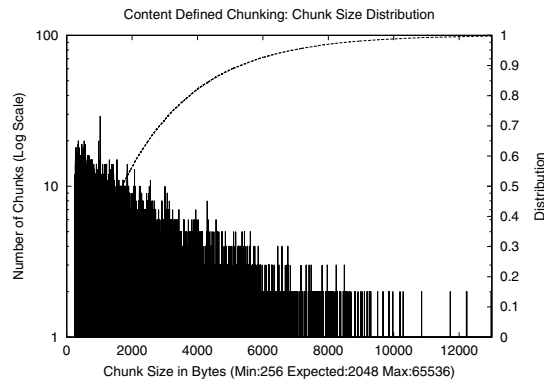


Figure 6: Emacs-20.7 CDC Distribution (Mean = 2 KB, Max = 64 KB). The left y-axis (log scale) corresponds to the histogram of chunk sizes, and the right y-axis shows the cumulative distribution.

the structure of the tree. As illustrated in Figure 5, the hash of a sub-directory D_S is computed by an in-order traversal of all its immediate children. For example, if $D_S = \{D_{S1}, D_{S2}\}$ then

$$h(D_S) = h(h(DN)h(D_{S1})h(UP)h(DN)h(D_{S2})h(UP))$$

where “UP” and “DN” are two literals representing the traversal of the up and down links in the tree respectively. Finally, the hash of the root node, D_R , of the directory tree is computed similar to that of a subtree defined above. The HHT algorithm, thus, outputs a list of the hash values of all the nodes, in the directory tree i.e., $h(D_R), h(D_A), h(D_S), \dots, h(D_L), \dots, h(f1) \dots$. Note that our HHT technique provides a hierarchical encoding of both the file content and the directory structure. This proves beneficial in eliminating directory trees identical in content and structure at the highest level.

The target, in turn, computes the HHT hash values of its directory tree and stores each element in a hash table. Each element of the HHT sent by the source—starting at the root node of the directory tree and if necessary progressing downward to the file nodes—is used to index into the target’s hash table to see if the node matches any node at the target. Thus, HHT finds the maximal common directory match and enables fast directory pruning since a match at any node implies that all the descendant nodes match as well. For example, if the root hash values match, then no further matching is done as the trees are identical in both content and structure. At the end of this phase, all exactly matching directory trees and files would have been pruned.

To illustrate the advantage of HHT, consider, for example, a rename update of the root directory of Linux Kernel 2.4.26 source tree. Even though no content was changed, rsync found no matching data and sent the entire tree of size 161.7 MB with an additional 1.03 MB of metadata (using the default block-size of 700 bytes). In

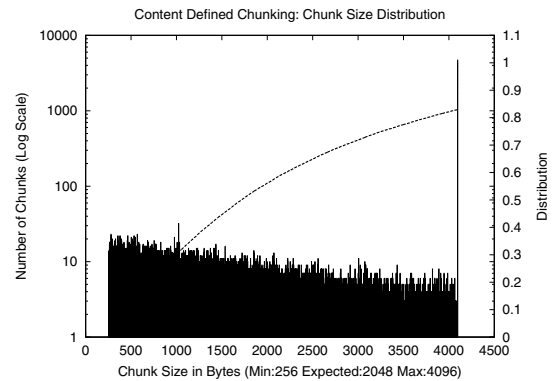


Figure 7: Emacs-20.7 CDC Distribution (Mean = 2KB, Max = 4 KB). The left y-axis (log scale) corresponds to the histogram of chunk sizes, and the right y-axis shows the cumulative distribution.

contrast, the HHT phase of TAPER sent 291 KB of the HHT metadata and determined, after a single hash match of the root node, that the entire data was identical.

The main advantages of using HHT for directory pruning are that it can: i) quickly (in $O(1)$ time) find the maximal exact match, ii) handle exact matches from the entire tree to individual files, iii) match both structure and content, and iv) handle file or directory renames and moves.

3.1.2 Matching Chunks

Once all the common files and directories have been eliminated, we are left with a set of unmatched files at the source and the target. In Phase II, to capture the data commonality across *all* files and further reduce the unmatched data, we rely on content-defined chunking (which we discussed in Section 2). During this phase, the target sends the SHA-1 hash values of the unique (to remove local redundancy) CDCs of all the remaining files to the source. Since CDC hashes can be indexed for fast matching, the source can quickly eliminate all the matching chunks across all the files between the source and target. The source stores the CDC hashes locally for re-use when synchronizing with multiple targets.

When using CDCs, two parameters—the expected chunk size and the maximum chunk size—have to be selected for a given workload. LBFS [20] used an expected chunk size of 8 KB with a maximum of 64 KB. The chunk sizes, however, could have a large variance around the mean. Figure 6 shows the frequency and cumulative distribution of chunk sizes for the Emacs-20.7 source tree using an expected chunk size value of 2 KB with no limitation on the chunk size except for the absolute maximum of 64 KB. As can be seen from the figure, the chunk sizes have a large variance, ranging from 256 bytes to 12 KB with a relatively long tail.

The maximum chunk size limits this variance by forcing a chunk to be created if the size exceeds the maxi-

imum value. However, a forced split at fixed size values makes the algorithm behave more like fixed-size block matching with poor resilience to updates. Figure 7 shows the distribution of chunk sizes for the same workload and expected chunk size value of 2 KB with a maximum value now set to 4 KB. Approximately 17% of the chunks were created due to this limitation.

Moreover, as an update affects the neighboring chunks, CDCs are not suited for fine-grained matches when there are small-sized updates sprinkled throughout the data. As we observed in Figures 2 and 3 in Section 2, CDC performed better than sliding-block for larger sized chunks, while rsync was better for finer-grained matches. We, therefore, use a relatively large expected chunk size (4 KB) in this phase to do fast, coarse-grained matching of data across all the remaining files. At the end of the chunk matching phase, the source has a set of files each with a sequence of matched and unmatched regions. In the next phase, doing finer-grained block matches, we try to reduce the size of these unmatched regions.

3.1.3 Matching Blocks

After the completion of the second phase, each file at the source would be in the form of a series of matched and unmatched regions. The contiguous unmatched chunks lying in-between two matched chunks of the same file are merged together and are called *holes*. To reduce the size of the holes, in this phase, we perform finer-grained block matching. The sliding-block match, however, can be applied only to a *pair* of files. We, therefore, need to determine the constituent files to match a pair of holes, i.e., we need to determine which pair of files at the source and target are similar. The technique we use for similarity detection is needed in multiple phases, hence, we discuss it in detail in Section 4. Once we identify the pair of similar files to compare, block matching is applied to the holes of the file at the source. We split the unmatched holes of a file, *f*, at the source using relatively smaller fixed-size blocks (700 bytes) and send the block signatures (Rabin fingerprint for weak rolling checksum; SHA-1 for strong checksum) to the target. At the target, a sliding-block match is used to compare against the holes in the corresponding file. The target then requests the set of unmatched blocks from the source.

To enable a finer-grained match, in this phase, the matching size of 700 bytes is selected to be a fraction of the expected chunk size of 4 KB. The extra cost of smaller blocks is offset by the fact that we have much less data (holes instead of files) to work with.

3.1.4 Matching Bytes

This final phase further reduces the bytes to be sent. After the third phase, the source has a set of unmatched blocks remaining. The source also has the set of matched

chunks, blocks and files that matched in the first three phases. To further reduce the bytes to be sent, the blocks in the unmatched set are *delta encoded* with respect to a similar block in the matched set. The target can then reconstruct the block by applying the delta-bytes to the matched block. Observe that unlike redundancy elimination techniques for storage, the source does not have the data at the target. To determine which matched and unmatched blocks are similar, we apply the similarity detection technique at the source.

Finally, the remaining unmatched blocks and the delta-bytes are further compressed using standard compression algorithms (e.g., gzip) and sent to the target. The data at the target is validated in the end by sending an additional checksum per file to avoid any inconsistencies.

3.1.5 Discussion

In essence, TAPER combines the faster matching of content-defined chunks and the finer matching of the sliding block approach. CDC helps in finding common data across all files, while sliding-block can find small random changes between a pair of files. Some of the issues in implementing TAPER require further discussion:

Phased refinement: The multiple phases of TAPER result in better differential compression. By using a coarse granularity for a larger dataset we reduce the metadata overhead. Since the dataset size reduces in each phase, it balances the computation and metadata overhead of finer granularity matching. The TAPER phases are not just recursive application of the same algorithm to smaller block sizes. Instead, they use the best approach for a particular size.

Re-using Hash computation: Unlike rsync where the source does the sliding-block match, TAPER stores the hash values at the source both in the directory matching and the chunk matching phase. These values need not be recomputed for different targets, thereby, increasing the scalability of TAPER. The hash values are computed either when the source file system is quiesced or over a consistent copy of the file system, and are stored in a local database.

Pluggable: The TAPER phases are pluggable in that some can be dropped if the desired level of data reduction has been achieved. For example, Phase I can be directly combined with Phase III and similarity detection giving us an rsync++. Another possibility is just dropping phases III and IV.

Round-trip latency: Each phase of TAPER requires a metadata exchange between the server and the target corresponding to one logical round-trip. This additional round-trip latency per phase is balanced by the fact that amount of data and metadata transferred is sufficiently reduced.

Hash collisions: In any hash-based differential compression technique there is the extremely low but non-zero probability of a hash collision [10]. In systems that use hash-based techniques to compress local data, a collision may corrupt the source file system. TAPER is used for replica synchronization and hence only affects the target data. Secondly, data is validated by a second cryptographic checksum over the entire file. The probability of two hash collisions over the same data is quadratically lower and we ignore that possibility.

The recent attack on the SHA-1 hash function [26] raises the challenge of an attacker deliberately creating two files with the same content [1]. This attack can be addressed by prepending a secret, known only to the root at the source and target, to each chunk before computing the hash value.

4 Similarity Detection

As we discussed in Section 3, the last two phases of the TAPER protocol rely on a mechanism for similarity detection. For block and byte matching, TAPER needs to determine which two files or chunks are similar. Similarity detection for files has been extensively studied in the WWW domain and relies on shingling [22] and super fingerprints discussed later in Section 4.3.

In TAPER, we explore the application of Bloom filters for file similarity detection. Bloom filters compactly represent a set of elements using a finite number of bits and are used to answer approximate set membership queries. Given that Bloom filters compactly represent a set, they can also be used to approximately match two sets. Bloom filters, however, cannot be used for exact matching as they have a finite false-match probability, but they are naturally suited for similarity matching. We first give a brief overview of Bloom filters, and later present and analyze the similarity detection technique.

4.1 Bloom Filters Overview

A Bloom filter is a space-efficient representation of a set. Given a set U , the Bloom filter of U is implemented as an array of m bits, initialized to 0 [4]. Each element u ($u \in U$) of the set is hashed using k independent hash functions h_1, \dots, h_k . Each hash function $h_i(u)$ for $1 \leq i \leq k$ returns a value between 1 and m then when an element is added to the set, it sets k bits, each bit corresponding to a hash function output, in the Bloom filter array to 1. If a bit was already set it stays 1. For set membership queries, Bloom filters may yield a *false positive*, where it may appear that an element v is in U even though it is not. From the analysis in the survey paper by Broder and Mitzenmacher [8], given $n = |U|$ and the Bloom filter size m , the optimal value of k that minimizes the false positive probability, p^k , where p denotes

that probability that a given bit is set in the Bloom filter, is $k = \frac{m}{n} \ln 2$.

4.2 Bloom Filters for Similarity Testing

Observe that we can view each file to be a set in Bloom filter parlance whose elements are the CDCs that it is composed of. Files with the same set of CDCs have the same Bloom filter representation. Correspondingly, files that are similar have a large number of 1s common among their Bloom filters. For multisets, we make each CDC unique before Bloom filter generation to differentiate multiple copies of the same CDC. This is achieved by attaching an index value of each CDC chunk to its SHA-1 hash. The index ranges from 1 to $\ln r$, where r is the multiplicity of the given chunk in the file.

For finding similar files, we compare the Bloom filter of a given file at the source with that of all the files at the replica. The file sharing the highest number of 1's (bit-wise AND) with the source file and above a certain threshold (say 70%) is marked as the matching file. In this case, the bit wise AND can also be perceived as the dot product of the two bit vectors. If the 1 bits in the Bloom filter of a file are a complete subset of that of another filter then it is highly probable that the file is included in the other.

Bloom filter when applied to similarity detection have several advantages. First, the compactness of Bloom filters is very attractive for remote replication (storage and transmission) systems where we want to minimize the metadata overheads. Second, Bloom filters enable fast comparison as matching is a bitwise-AND operation. Third, since Bloom filters are a complete representation of a set rather than a deterministic sample (e.g., shingling), they can determine inclusions effectively e.g., tar files and libraries. Finally, as they have a low metadata overhead they could be combined further with either sliding block or CDC for narrowing the match space.

To demonstrate the effectiveness of Bloom filters for similarity detection, consider, for example, the file *ChangeLog* in the Emacs-20.7 source distribution which we compare against all the remaining 1967 files in the Emacs-20.1 source tree. 119 identical files out of a total 2086 files were removed in the HHT phase. The CDCs of the files were computed using an expected and maximum chunk size of 1 KB and 2 KB respectively. Figure 8 shows that the corresponding *ChangeLog* file in the Emacs-20.1 tree matched the most with about 90% of the bits matching.

As another example, consider the file *nt/config.nt* in Emacs-20.7 (Figure 9) which we compare against the files of Emacs-20.1. Surprisingly, the file that matched most was *src/config.in*—a file with a different name in a different directory tree. The CDC expected and maximum chunk sizes were 512 bytes and 1 KB respec-

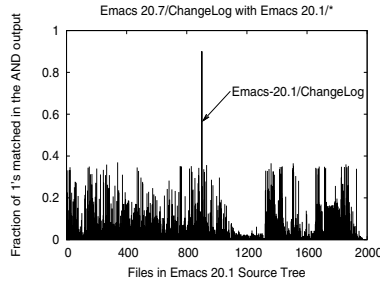


Figure 8: Bloom filter Comparison of the file 'Emacs-20.7/ChangeLog' with files 'Emacs-20.1/*'

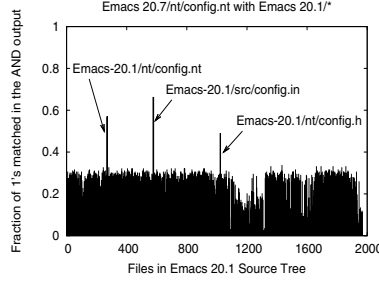


Figure 9: Bloom filter Comparison of the file 'Emacs-20.7/nt/config.nt' with files 'Emacs-20.1/*'

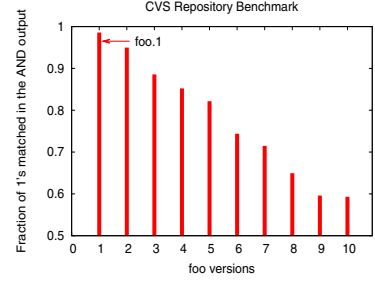


Figure 10: Bloom filter Comparison of file 'foo' with later versions 'foo.1', 'foo.2', ... 'foo.10'

tively. Figure 9 shows that while the file with the same name *nt/config.nt* matched in 57% of the bits, the file *src/config.in* matched in 66%. We further verified this by computing the corresponding *diff* output of 1481 and 1172 bytes, respectively. This experiment further emphasizes the need for content-based similarity detection.

To further illustrate that Bloom filters can differentiate between *multiple* similar files, we extracted a technical documentation file 'foo' (say) (of size 175 KB) incrementally from a CVS archive, generating 10 different versions, with 'foo' being the original, 'foo.1' being the first version (with a change of 4154 bytes from 'foo') and 'foo.10' being the last. The CDC chunk sizes were chosen as in the *ChangeLog* file example above. As shown in Figure 10, the Bloom filter for 'foo' matched the most (98%) with the closest version 'foo.1' and the least (58%) with the latest version 'foo.10'.

4.2.1 Analysis

The main consideration when using Bloom filters for similarity detection is the false match probability of the above algorithm as a function of similarity between the source and a candidate file. Extending the analysis for membership testing [4] to similarity detection, we proceed to determine the expected number of *inferred* matches between the two sets. Let A and B be the two sets being compared for similarity. Let m denote the number of bits (size) in the Bloom filter. For simplicity, assume that both sets have the same number of elements. Let n denote the number of elements in both sets A and B i.e., $|A| = |B| = n$. As before, k denotes the number of hash functions. The probability that a bit is set by a hash function h_i for $1 \leq i \leq k$ is $\frac{1}{m}$. A bit can be set by any of the k hash functions for each of the n elements. Therefore, the probability that a bit is not set by any hash function for any element is $(1 - \frac{1}{m})^{nk}$. Thus, the probability, p , that a given bit is set in the Bloom filter of A is given by:

$$p = \left(1 - \left(1 - \frac{1}{m}\right)^{nk}\right) \approx 1 - e^{-\frac{nk}{m}} \quad (1)$$

For an element to be considered a member of the set, all the corresponding k bits should be set. Thus, the probability of a false match, i.e., an outside element is inferred as being in set A , is p^k . Let C denote the intersection of sets A and B and c denote its cardinality, i.e., $C = A \cap B$ and $|C| = c$.

For similarity comparison, let us take each element in set B and check if it belongs to the Bloom filter of the given set A . We should find that the c common elements will definitely match and a few of the other $(n - c)$ may also match due to the false match probability. By Linearity of Expectation, the expected number of elements of B inferred to have matched with A is

$$E[\text{\# of inferred matches}] = (c) + (n - c)p^k$$

To minimize the false matches, this expected number should be as close to c as possible. For that $(n - c)p^k$ should be close to 0, i.e., p^k should approach 0. This happens to be the same as minimizing the probability of a false positive. Expanding p and under asymptotic analysis, it reduces to minimizing $(1 - e^{-\frac{nk}{m}})^k$. Using the same analysis for minimizing the false positive rate [8], the minima obtained after differentiation is when $k = \frac{m}{n} \ln 2$. Thus, the expected number of inferred matches for this value of k becomes

$$E[\text{\# of inferred matches}] = c + (n - c)(0.6185)^{\frac{m}{n}}$$

Thus, the expected number of bits set corresponding to inferred matches is

$$E[\text{\# of matched bits}] = m \left[1 - \left(1 - \frac{1}{m}\right)^{k(c + (n-c)(0.6185)^{\frac{m}{n}})} \right]$$

Under the assumption of perfectly random hash functions, the expected number of total bits set in the Bloom filter of the source set A , is mp . The ratio, then, of the expected number of matched bits corresponding to inferred matches in $A \cap B$ to the expected total number of bits set in the Bloom filter of A is:

$$\frac{E[\text{\# of matched bits}]}{E[\text{\# total bits set}]} = \frac{\left(1 - e^{-\frac{k}{m}(c + (n-c)(0.6185)^{\frac{m}{n}})}\right)}{\left(1 - e^{-\frac{nk}{m}}\right)}$$

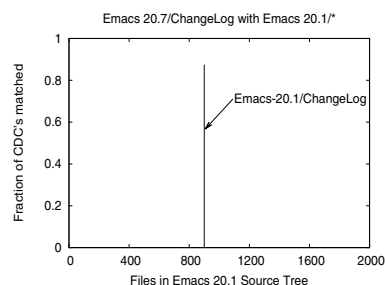


Figure 11: CDC comparison of the file 'Emacs-20.7/ChangeLog' with files 'Emacs-20.1/*'

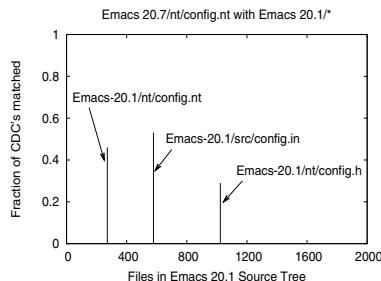


Figure 12: CDC comparison of the file 'Emacs-20.7/nt/config.nt' with files 'Emacs-20.1/*'

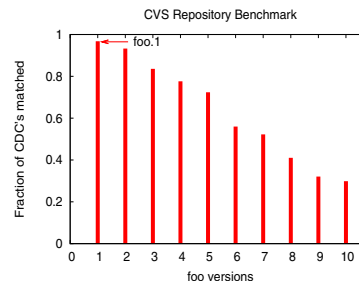


Figure 13: CDC Comparison of file 'foo' with later versions 'foo.1', 'foo.2', ... 'foo.10'

Observe that this ratio equals 1 when all the elements match, i.e., $c = n$. If there are no matching elements, i.e., $c = 0$, the ratio $= 2(1 - (0.5)^{(0.6185)\frac{m}{n}})$. For $m = n$, this evaluates to 0.6973, i.e., 69% of matching bits may be false. For larger values, $m = 2n, 4n, 8n, 10n, 11n$, the corresponding ratios are 0.4658, 0.1929, 0.0295, 0.0113, 0.0070 respectively. Thus, for $m = 11n$, on an average, less than 1% of the bits set may match incorrectly. The expected ratio of matching bits is highly correlated to the expected ratio of matching elements. Thus, if a large fraction of the bits match, then it's highly likely that a large fraction of the elements are common.

Although the above analysis was done based on expected values, we show in an extended technical report [13] that under the assumption that the difference between p and $(1 - e^{-\frac{nk}{m}})$ is very small, the *actual* number of matched bits is highly concentrated around the *expected* number of matched bits with small variance [18].

Given that the number of bits in the Bloom filter should be larger than the number of elements in the set we need large filters for large files. One approach is to select a new filter size when the file size doubles and only compare the files represented with the same filter size. To support subset matching, however, the filter size for all the files should be identical and therefore all files need to have a filter size equal to size required for the largest file.

4.2.2 Size of the Bloom Filter

As discussed in the analysis, the fraction of bits matching incorrectly depends on the size of the Bloom filter. For a 97% accurate match, the number of bits in the Bloom filter should be 8x the number of elements (chunks) in the set (file). For a file of size 128 KB, an expected and maximum chunk size of 4 KB and 64 KB, respectively results in around 32 chunks. The Bloom filter is set to be 8x this value i.e., 256 bits. For small files, we can set the expected chunk size to 256 bytes. Therefore, the Bloom filter size is set to 8x the expected number of chunks (32 for 8 KB file) i.e., 256 bits, which is a 0.39% and 0.02% overhead for file size of 8 KB and 128 KB, respectively.

4.3 Comparison with Shingling

Previous work on file similarity has mostly been based on shingling or super fingerprints. Using this method, for each object, all the k consecutive words of a file (called k -shingles) are hashed using Rabin fingerprint [22] to create a set of fingerprints (also called features or pre-images). These fingerprints are then sampled to compute a super-fingerprint of the file. Many variants have been proposed that use different techniques on how the shingle fingerprints are sampled (min-hashing, Mod_m , Min_s , etc.) and matched [5–7]. While Mod_m selects all fingerprints whose value modulo m is zero; Min_s selects the set of s fingerprints with the smallest value. The min-hashing approach further refines the sampling to be the min values of say 84 random min-wise independent permutations (or hashes) of the set of all shingle fingerprints. This results in a fixed size sample of 84 fingerprints that is the resulting feature vector. To further simplify matching, these 84 fingerprints can be grouped as 6 “super-shingles” by concatenating 14 adjacent fingerprints [9]. In REBL [15] these are called super-fingerprints. A pair of objects are then considered similar if either all or a large fraction of the values in the super-fingerprints match.

Our Bloom filter based similarity detection differs from the shingling technique in several ways. It should be noted, however, that the variants of shingling discussed above improve upon the original approach and we provide a comparison of our technique with these variants wherever applicable. First, shingling (Mod_m , Min_s) computes file similarity using the intersection of the two feature sets. In our approach, it requires only the bit-wise AND of the two Bloom filters (e.g., two 128 bit vectors). Next, shingling has a higher computational overhead as it first segments the file into k -word shingles ($k = 5$ in [9]) resulting in shingle set size of about $S - k + 1$, where S is the file size. Later, it computes the image (value) of each shingle by applying set (say H) of min-wise independent hash functions ($|H|=84$ [9]) and then for each function, selecting the shingle corre-

sponding to the minimum image. On the other hand, we apply a set of independent hash functions (typically less than 8) to the chunk set of size on average $\lceil \frac{S}{c} \rceil$ where c is the expected chunk size (e.g., $c=256$ bytes for $S=8$ KB file). Third, the size of the feature set (number of shingles) depends on the sampling technique in shingling. For example, in Mod_m , even some large files might have very few features whereas small files might have zero features. Some shingling variants (e.g., Min_s , Mod_{2i}) aim to select roughly a constant number of features. Our CDC based approach only varies the chunk size c , to determine the number of chunks as a trade-off between performance and fine-grained matching. We leave the empirical comparison with shingling as future work. In general, a compact Bloom filter is easier to attach as a file tag and is compared simply by matching the bits.

4.4 Direct Chunk Matching for Similarity

The chunk-based matching in the second phase, can be directly used to simultaneously detect similar files between the source and target. When matching the chunk hashes belonging to a file, we create a list of candidate files that have a common chunk with the file. The file with the maximum number of matching chunks is marked as the similar file. Thus the matching complexity of direct chunk matching is $O(\text{Number of Chunks})$. This direct matching technique can also be used in conjunction with other similarity detection techniques for validation. While the Bloom filter technique is general and can be applied even when a database of all file chunks is not maintained, direct matching is a simple extension of the chunk matching phase.

To evaluate the effectiveness of similarity detection using CDC, we perform the same set of experiments as discussed in Section 4.2 for Bloom filters. The results, as expected, were identical to the Bloom filter approach. Figures 11, 12, and 13 show the corresponding plots for matching the files 'ChangeLog', 'nt/config.nt' and 'foo', respectively. Direct matching is more exact as there is no probability of false matching. The Emacs-20.1/ChangeLog file matched with the Emacs-20.7/ChangeLog file in 112 out of 128 CDCs (88%). Similarly, the Emacs-20.7/nt/config.nt file had a non-zero match with only three Emacs-20.1/* files with 8 (46%), 9 (53%), 5 (29%) matches out of 17 corresponding to the files nt/config.nt, src/config.in and nt/config.h, resp. The file 'foo' matched 'foo.1' in 99% of the CDCs.

5 Experimental Evaluation

In this section, we evaluate TAPER using several workloads, analyze the behavior of the various phases of the protocol and compare the bandwidth efficiency, computation overhead, and response times with tar+gzip, rsync, and CDC.

5.1 Methodology

We have implemented a prototype of TAPER in C and Perl. The chunk matching in Phase II uses code from the CDC implementation of LBFS [20] and uses the SleepyCat software's BerkeleyDB database package for providing hash based indexing. The delta-compression of Phase IV was implemented using vcdiff [14]. The experimental testbed used two 933 MHz Intel Pentium III workstations with 512 MB of RAM running Linux kernel 2.4.22 connected by full-duplex 100 Mbit Ethernet.

Software Sources (Size KB)		
Workload	No. of Files	Total Size
<i>linux-src (2.4.26)</i>	13235	161,973
<i>AIX-src (5.3)</i>	36007	874,579
<i>emacs (20.7)</i>	2086	54,667
<i>gcc (3.4.1)</i>	22834	172,310
<i>rsync (2.6.2)</i>	250	7,479
Object Binaries (Size MB)		
<i>linux-bin (Fedora)</i>	38387	1,339
<i>AIX-bin (5.3)</i>	61527	3,704
Web Data (Size MB)		
<i>CNN</i>	13534	247
<i>Yahoo</i>	12167	208
<i>IBM</i>	9223	248
<i>Google Groups</i>	16284	251

Table 1: Characteristics of the different Datasets

For our analysis, we used three different kinds of workloads: i) software distribution sources, ii) operating system object binaries, and iii) web content. Table 1 details the different workload characteristics giving the total uncompressed size and the number of files for the newer version of the data at the source.

Workload	<i>linux-src</i>	<i>AIX-src</i>	<i>emacs</i>	<i>gcc</i>
Versions	2.4.26 - 2.4.22	5.3 - 5.2	20.7 - 20.1	3.4.1 - 3.3.1
Size KB	161,973	874,579	54,667	172,310
Phase I	62,804	809,514	47,954	153,649
Phase II	24,321	302,529	30,718	98,428
Phase III	20,689	212,351	27,895	82,952
Phase IV	18,127	189,528	26,126	73,263
Diff Output	10,260	158,463	14,362	60,215

Table 2: Evaluation of TAPER Phases. The numbers denote the unmatched data in KB remaining at the end of a phase.

Software distributions sources For the software distribution workload, we consider the source trees of the gcc compiler, the emacs editor, rsync, the Linux kernel, and the AIX kernel. The data in the source trees consists of only ASCII text files. The gcc workload represents the source tree for GNU gcc versions 3.3.1 at the targets and version 3.4.1 at the source. The emacs dataset con-

sists of the source code for GNU Emacs versions 20.1 and 20.7. Similarly, the *rsync* dataset denotes the source code for the rsync software versions 2.5.1 and 2.6.2, with the addition that 2.6.2 also includes the object code binaries of the source. The two kernel workloads, *linux-src* and *AIX-src*, comprise the source tree of the Linux kernel versions 2.4.22 and 2.4.26 and the source tree for the AIX operating system versions 5.2 and 5.3, respectively.

Object binaries Another type of data widely upgraded and replicated is code binaries. Binary files have different characteristics compared to ASCII files. To capture a tree of code binaries, we used the operating system binaries of Linux and AIX. We scanned the entire contents of the directory trees */usr/bin*, */usr/X11R6* and */usr/lib* in RedHat 7.3 and RedHat Fedora Core I distributions, denoted by *linux-bin* dataset. The majority of data in these trees comprises of system binaries and software libraries containing many object files. The *AIX-bin* dataset consists of object binaries and libraries in */usr*, */etc*, */var*, and */sbin* directories of AIX versions 5.2 and 5.3.

Web content Web data is a rich collection of text, images, video, binaries, and various other document formats. To get a representative sample of web content that can be replicated at mirror sites and CDNs, we used a web crawler to crawl a number of large web servers. For this, we used the *wget* 1.8.2 crawler to retrieve the web pages and all the files linked from them, recursively for an unlimited depth. However, we limited the size of the downloaded content to be 250 MB and restricted the crawler to remain within the website's domain.

The four datasets, CNN, Yahoo, IBM and Google Groups, denote the content of *www.cnn.com*, *www.yahoo.com*, *www.ibm.com*, and *groups.google.com* websites that was downloaded every day from 15 Sep. to 10 Oct., 2004. CNN is a news and current affairs site wherein the top-level web pages change significantly over a period of about a day. Yahoo, a popular portal on the Internet, represents multiple pages which have small changes corresponding to daily updates. IBM is the company's corporate homepage providing information about its products and services. Here, again the top-level pages change with announcements of product launches and technology events, while the others relating to technical specifications are unchanged. For the Google Groups data set, most pages have numerous changes due to new user postings and updates corresponding to feedback and replies.

5.2 Evaluating TAPER Phases

As we described earlier, TAPER is a multi-phase protocol where each phase operates at a different granularity. In this section, we evaluate the behavior of each phase on different workloads. For each dataset, we upgrade the

Workload	<i>linux-src</i>	<i>AIX-src</i>	<i>emacs</i>	<i>gcc</i>
Phase I	291	792	46	502
Phase II	317	3,968	241	762
Phase III	297	3,681	381	1,204

Table 3: Uncompressed Metadata overhead in KB of the first three TAPER phases.

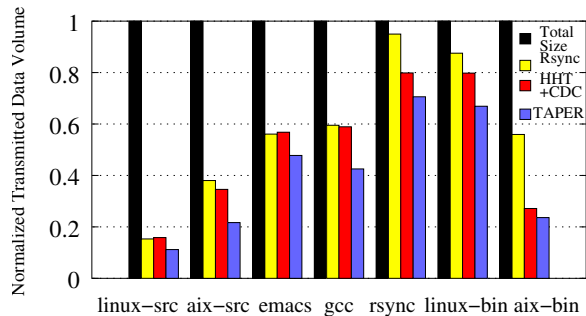


Figure 14: Normalized transmitted data volume (uncompressed) by Rsync, HHT+CDC, TAPER on Software distribution and Object binaries. The results are normalized against the total size of the dataset.

older version to the newer version, e.g., Linux version 2.4.22 to 2.4.26. For each phase, we measure the total size of unmatched data that remains for the next phase and the total metadata that was exchanged between the source and the target. The parameters used for expected and max chunk size in Phase II was 4 KB and 64 KB, respectively. For Phase III, the block size parameter was 700 bytes. The data for Phase IV represents the final unmatched data that includes the delta-bytes. In practice, this data would then be compressed using *gzip* and sent to the target. We do not present the final compressed numbers here as we want to focus on the contribution of TAPER and not *gzip*. For comparison, we show the size of the output of “diff -r”. Table 2 shows the total unmatched data that remains after the completion of a phase for the workloads *linux-src*, *AIX-src*, *emacs* and *gcc*. Additionally, Table 3 shows the metadata that was transmitted for each phase for the same workloads. The table shows that the data reduction in terms of uncompressed bytes transmitted range from 88.8% for the *linux-src* and 78.3% for the *AIX-src* to 52.2% for *emacs* and 58% for *gcc*. On the other hand, the overhead (compared to the original data) of metadata transmission ranged from 0.5% for *linux-src* and 0.9% for *AIX-src* to 1.2% for *emacs* and 1.4% for *gcc*. Observe that the metadata in Phase II and III is in the same ball park although the matching granularity is reduced by an order of magnitude. This is due to the unmatched data reduction per phase. The metadata overhead of Phase I is relatively high. This is partly due to the strong 20-byte hash SHA-1 hash that is used. Note that the unmatched data at the end of Phase IV is in the

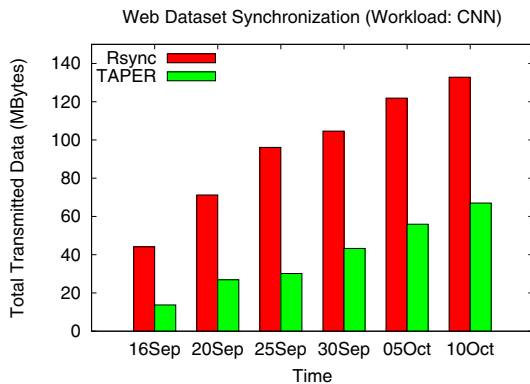


Figure 15: Rsync, TAPER Comparison on CNN web dataset

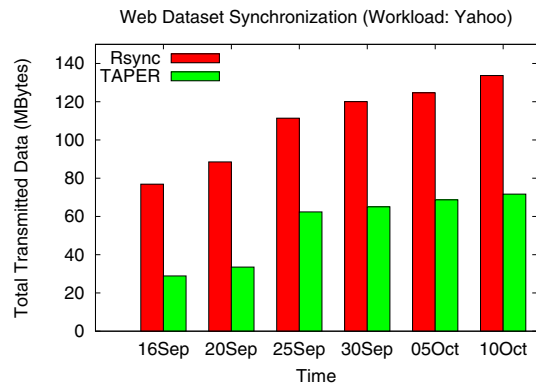


Figure 16: Rsync, TAPER Comparison on Yahoo web dataset

same ball park as the *diff* output between the new and old data version but that computing the latter requires a node to have a copy of both versions and so is not a viable solution to our problem.

5.3 Comparing Bandwidth Efficiency

In this section, we compare the bandwidth efficiency of TAPER (in terms of total data and metadata transferred) with tar+gzip, rsync, and HHT+CDC. To differentiate bandwidth savings due to TAPER from data compression (gzip), we first illustrate TAPER's contribution to bandwidth savings without gzip for software sources and object binaries workloads. Figure 14 shows the normalized transmitted data volume by TAPER, rsync, and HHT+CDC for the given datasets. The transmitted data volume is normalized against the total size of the dataset. For the *gcc*, *AIX-src*, and *linux-bin* datasets, rsync transmitted about 102 MB, 332 MB, and 1.17 GB, respectively. In comparison, TAPER sent about 73 MB, 189 MB, and 896 MB corresponding to bandwidth savings of 29%, 43% and 24%, respectively for these three datasets. Overall, we observe that TAPER's improvement over rsync ranged from 15% to 43% for software sources and 24% to 58% for object binaries workload.

Using gzip compression, we compare TAPER and rsync with the baseline technique of *tar+gzip*. For the *linux-src* and *AIX-bin* data-sets, the compressed tarball (tar+gzip) of the directory trees, Linux 2.4.26 and AIX 5.3, are about 38 MB and 1.26 GB, respectively. TAPER (with compression in the last phase) sent about 5 MB and 542 MB of difference data, i.e., a performance gain of 86% and 57% respectively over the compressed tar output. Compared to rsync, TAPER's improvement ranged from 18% to 25% for software sources and 32% to 39% for object binaries datasets.

For web datasets, we marked the data crawled on Sep. 15, 2004 as the base set and the six additional versions corresponding to the data gathered after 1, 5, 10, 15, 20

and 25 days. We examined the bandwidth cost of updating the base set to each of the updated versions without compression. Figures 15, 16, 17, 18 show the total data transmitted (without compression) by TAPER and rsync to update the base version for the web datasets. For the CNN workload, the data transmitted by TAPER across the different days ranged from 14 MB to 67 MB, while that by rsync ranged from 44 MB to 133 MB. For this dataset, TAPER improved over rsync from 54% to 71% without compression and 21% to 43% with compression. Similarly, for the Yahoo, IBM and Google groups workload, TAPER's improvement over rsync without compression ranged 44-62%, 26-56%, and 10-32%, respectively. With compression, the corresponding bandwidth savings by TAPER for these three workloads ranged 31-57%, 23-38%, and 12-19%, respectively.

5.4 Comparing Computational Overhead

In this section, we evaluate the overall computation overhead at the source machine. Micro-benchmark experiments to analyze the performance of the individual phases are given in Section 5.5. Intuitively, a higher computational load at the source would limit its scalability.

For the emacs dataset, the compressed tarball takes 10.4s of user and 0.64s of system CPU time. The corresponding CPU times for rsync are 14.32s and 1.51s. Recall that the first two phases of TAPER need only to be computed once and stored. The total CPU times for the first two phases are 13.66s (user) and 0.88s (system). The corresponding total times for all four phases are 23.64s and 4.31s. Thus, the target specific computation only requires roughly 13.5s which is roughly same as rsync. Due to space constraints, we omit these results for the other data sets, but the comparisons between rsync and TAPER are qualitatively similar for all experiments.

5.5 Analyzing Response Times

In this section, we analyze the response times for the various phases of TAPER. Since the phases of TAPER

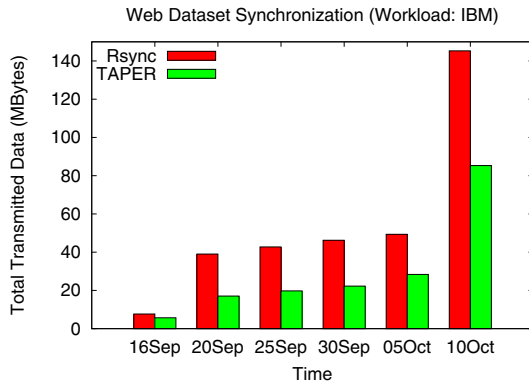


Figure 17: Rsync, TAPER Comparison on IBM web dataset

Chunk Sizes File Size	256 Bytes (ms)	512 Bytes (ms)	2 KB (ms)	8 KB (ms)
100 KB	4	3	3	2
1 MB	29	27	26	24
10 MB	405	321	267	259

Table 4: CDC hash computation time for different files and expected chunk sizes

include sliding-block and CDC, the same analysis holds for rsync and any CDC-based system. The total response time includes the time for i) hash-computation, ii) matching, iii) metadata exchange, and iv) final data transmission. In the previous discussion on bandwidth efficiency, the total metadata exchange and data transmission byte values are a good indicator of the time spent in these two components. The other two components of hash-computation and matching are what we compare next.

The hash-computation time for a single block, used in the sliding-block phase, to compute a 2-byte checksum and a 4-byte MD4 hash for block sizes of 512 bytes, 2 KB, and 8 KB, are $5.37\mu s$, $19.77\mu s$, and $77.71\mu s$, respectively. Each value is an average of 1000 runs of the experiment. For CDC, the hash-computation time includes detecting the chunk boundary, computing the 20-byte SHA-1 signature and populating the database for indexing. Table 4 shows the CDC computation times for different file sizes of 100 KB, 1 MB, and 10 MB, using different expected chunk sizes of 256 bytes, 512 bytes, 2 KB, and 8 KB, respectively. The Bloom filter generation time for a 100 KB file (309 CDCs) takes 118ms, 120ms, and 126ms for 2, 4, and 8 hash functions, respectively.

Figure 19 shows the match time for sliding-block and CDC for the 3 file sizes (10 KB, 1 MB and 10 MB) and 3 block sizes (512 bytes, 2 KB, 8 KB). Although the fixed-block hash generation is 2 to 4 times faster than CDC chunk hash-computation, the time for CDC matching is 10 to a 100 times faster. The hash-computation time can be amortized over multiple targets as the results

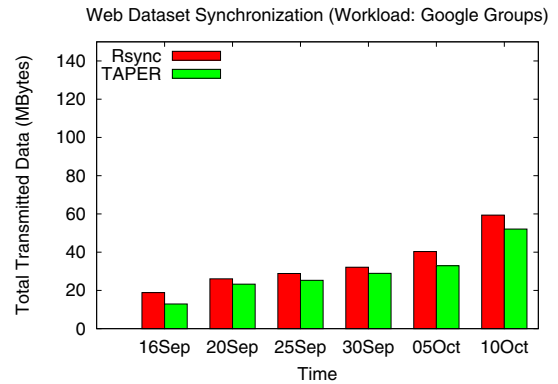


Figure 18: Rsync, TAPER Comparison on Google Groups web dataset

are stored in a database and re-used. Since the matching time is much faster for CDC we use it in Phase II where it is used to match all the chunks over all the files.

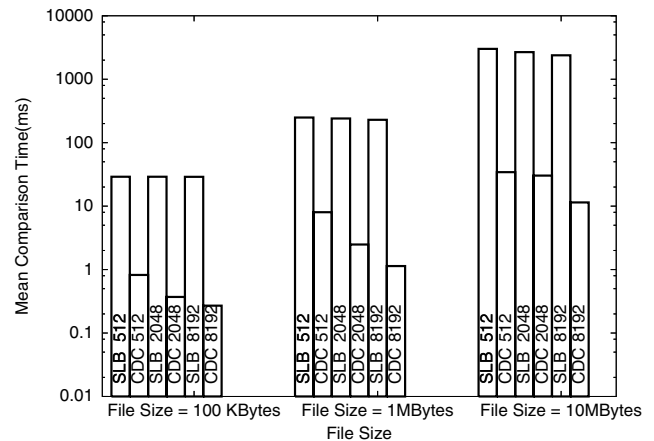


Figure 19: Matching times for CDC and sliding-block (SLB).

6 Related Work

Our work is closely related to two previous hash-based techniques: sliding block used in rsync [2], and CDC introduced in LBFS [20]. As discussed in Section 2, the sliding-block technique works well only under certain conditions: small file content updates but no directory structure changes (renames, moves, etc.). Rsync uses sliding block only and thus performs poorly in name-resilience, scalability, and matching time. TAPER, however, uses sliding block in the third phase when these conditions hold. The CDC approach, in turn, is sensitive to the chunk size parameter: small size leads to fine-grained matching but high metadata whereas large chunk size results in lower metadata but fewer matches. Some recent studies have proposed multiresolution partitioning of data blocks to address the problem of the optimal block-size both in the context of rsync [16] and CDC [12]. This results in a trade-off between bandwidth

savings and the number of network round-trips.

Previous efforts have also explored hash-based schemes based on sliding block and CDC for duplicate data suppression in different contexts. Mogul et al. use MD5 checksums over web payloads to eliminate redundant data transfers over HTTP links [19]. Rhea et al. describe a CDC based technique that removes duplicate payload transfers at finer granularities [23] compared to Mogul's approach. Venti [21] uses cryptographic hashes on CDCs to reduce duplication in an archival storage system. Farsite [3], a secure, scalable distributed file system, employs file level hashing to reclaim storage space from duplicate files. You et al. examine whole-file hashing and CDCs to suppress duplicate data in the Deepstore archival storage system [27]. Sapuntzakis et al. compute SHA-1 hashes of files to reduce data transferred during the migration of appliance states between machines [24].

For similarity detection, Manber [17] originally proposed the shingling technique to find similar files in a large file system. Broder refined Manber's technique by first using a deterministic sample of the hash values (e.g., min-hashing) and then coalescing multiple sampled fingerprints into *super-fingerprints* [5–7]. In contrast, TAPER uses Bloom filters [4] which compactly encode the CDCs of a given file to save bandwidth and performs fast bit-wise AND of Bloom filters for similarity detection. Bloom filters have been proposed to estimate the cardinality of set intersection [8] but have never been applied for near-duplicate elimination in file systems. Further improvements on Bloom filters can be achieved by using compressed Bloom filters [18], which reduce the number of bits transmitted over the network at the cost of increasing storage and computation costs.

7 Conclusion

In this paper we present TAPER, a scalable data replication protocol for replica synchronization that provides four redundancy elimination phases to balance the trade-off between bandwidth savings and computation overheads. Experimental results show that in comparison with rsync, TAPER reduces bandwidth savings by 15% to 71%, performs faster matching, and scales to a larger number of replicas. In future work, instead of synchronizing data on a per-client basis, TAPER can (a) use multicast to transfer the common updates to majority of the clients, and later (b) use cooperative synchronization where clients exchange small updates among themselves for the remaining individual differences.

8 Acknowledgments

We thank Rezaul Chowdhury, Greg Plaxton, Sridhar Rajagopalan, Madhukar Korupolu, and the anonymous reviewers for their insightful comments. This work was done while the first author was an intern at IBM Almaden Research Center. This work was supported in part by the

NSF (CNS-0411026), the Texas Advanced Technology Program (003658-0503-2003), and an IBM Faculty Research Award.

References

- [1] <http://th.informatik.uni-mannheim.de/people/lucks/hashcollisions/>.
- [2] Rsync <http://rsync.samba.org>.
- [3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, Dec. 2002.
- [4] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [5] A. Z. Broder. On the resemblance and containment of documents. In *SEQUENCES*, 1997.
- [6] A. Z. Broder. Identifying and filtering near-duplicate documents. In *COM*, pages 1–10, 2000.
- [7] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *WWW*, 1997.
- [8] A. Z. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. In *Allerton*, 2002.
- [9] D. Fetterly, M. Manasse, and M. Najork. On the evolution of clusters of near-duplicate web pages. In *LA-WEB*, 2003.
- [10] V. Henson. An analysis of compare-by-hash. In *HotOS IX*, 2003.
- [11] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. Technical Report TR-3002, Network Appliance Inc.
- [12] U. Imrak and T. Suel. Hierarchical substring caching for efficient content distribution to low-bandwidth clients. In *WWW*, 2005.
- [13] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered approach for eliminating redundancy in replica synchronization. Technical Report TR-05-42, Dept. of Comp. Sc., Univ. of Texas at Austin.
- [14] D. G. Korn and K.-P. Vo. Engineering a differencing and compression data format. In *USENIX Annual Technical Conference, General Track*, pages 219–228, 2002.
- [15] P. Kulkarni, F. Douglass, J. D. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2004.
- [16] J. Langford. Multiround rsync. Unpublished manuscript. <http://www-2.cs.cmu.edu/~jcl/research/mrsync/mrsync.ps>.
- [17] U. Manber. Finding similar files in a large file system. In *USENIX Winter Technical Conference*, 1994.
- [18] M. Mitzenmacher. Compressed Bloom filters. *IEEE/ACM Trans. Netw.*, 10(5):604–612, 2002.
- [19] J. C. Mogul, Y.-M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *NSDI*, 2004.
- [20] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *SOSP*, 2001.
- [21] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *FAST*, 2002.
- [22] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard University, 1981.
- [23] S. C. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *WWW*, pages 619–628, 2003.
- [24] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *OSDI*, Dec. 2002.
- [25] R. Thurlow. A server-to-server replication/migration protocol. IETF Draft May 2003.
- [26] X. Wang, Y. Yin, and H. Yu. Finding collisions in the full SHA1. In *Crypto*, 2005.
- [27] L. You, K. Pollack, and D. D. E. Long. Deep store: an archival storage system architecture. In *ICDE*, pages 804–815, 2005.

markable resistance to change ever since the IBM PC first jump-started personal computing. As the architecture timeline in Figure 2 illustrates, the persistently dominant x86 architecture has experienced only a few major architectural changes during its lifetime—32-bit registers and addressing in 1985, vector processing upgrades starting in 1996, and 64-bit registers and addressing in 2003. More importantly, each of these upgrades has religiously preserved backward code compatibility. Of the other architectures introduced during this period, none have come close to displacing the x86 architecture in the mainstream.

From these facts we observe that *instruction encodings are historically more durable than data encodings*. We will still be able to run x86 code efficiently decades from now, but it is less likely that future operating systems and applications will still include robust, actively-maintained decoders for today's compressed data streams.

1.1 Virtualizing Decoders

Virtual eXecutable Archives, or VXA, is a novel archival storage architecture that preserves data usability by packaging executable x86-based decoders along with compressed content. These decoders run in a specialized virtual machine (VM) that minimizes dependence on evolving host operating systems and processors. VXA decoders run on a well-defined subset of the unprivileged 32-bit x86 instruction set, and have no direct access to host OS services. A decoder only extracts archived data into simpler, and thus hopefully more “future-proof,” uncompressed formats: decoders cannot have user interfaces, open arbitrary files, or communicate with other processes.

By building on the ubiquitous native x86 architecture instead of using a specialized abstract machine such as Lorie's archival “Universal Virtual Computer” [27], VXA enables easy re-use of existing decoders written in arbitrary languages such as C and assembly language, which can be built with familiar development tools such as GCC. Use of the x86 architecture also makes execution of virtualized decoders extremely efficient on x86-based host machines, which is important to the many popular “short-term” uses of archives such as backups, software distribution, and structured document compression. VXA permits decoders access to the x86 vector processing instructions, further enhancing the performance of multimedia codecs.

Besides preserving long-term data usability, the VXA virtual machine also isolates the host system from buggy or malicious decoders. Decoder security vulnerabilities, such as the recent critical JPEG bug [31], cannot compromise the host under VXA. This security benefit is important because data decoders tend to be inherently

complex and difficult to validate, they are frequently exposed to data arriving from untrusted sources such as the Web, and they are usually perceived as too low-level and performance-critical to be written in type-safe languages.

1.2 Prototype Implementation

A prototype implementation of the VXA architecture, vxZIP/vxUnZIP, extends the well-known ZIP/UnZIP archive tools with support for virtualized decoders. The vxZIP archiver can attach VXA decoders both to files it compresses and to input files already compressed with recognized lossy or lossless algorithms. The vxUnZIP archive reader runs these VXA decoders to extract compressed files. Besides enhancing the durability of ZIP files themselves, vxZIP thus also enhances the durability of pre-compressed data stored in ZIP files, and can evolve to employ the latest specialized compression schemes without restricting the usability of the resulting archives.

VXA decoders stored in vxZIP archives are themselves compressed using a fixed algorithm (the “deflate” method standard for existing ZIP files) to reduce their storage overhead. The vxZIP prototype currently includes six decoders for both general-purpose data and specialized multimedia streams, ranging from 26 to 130KB in compressed size. Though this storage overhead may be significant for small archives, it is usually negligible for larger archives in which many files share the same decoder.

The prototype vxZIP/vxUnZIP tools run on both the 32-bit and 64-bit variants of the x86 architecture, and rely only on unprivileged facilities available on any mature x86 operating system. The performance cost of virtualization, compared with native x86-32 execution, is between 0 and 11% measured across six widely-available general-purpose and multimedia codecs. The cost is somewhat higher, 8–31%, compared with native x86-64 execution, but this difference is due not to virtualization overhead but to the fact that VXA decoders are always 32-bit, and thus cannot take advantage of the new 64-bit instruction set. The virtual machine that vxUnZIP uses to run the archived decoders is also available as a standalone library, which can be re-used to implement virtualization and isolation of extension modules for other applications.

Section 2 of this paper first presents the VXA architecture in detail. Section 3 then describes the prototype vxZIP/vxUnZIP tools, and Section 4 details the virtual machine monitor in which vxUnZIP runs archived decoders. Section 5 evaluates the performance and storage costs of the virtualized decoders. Finally, Section 6 summarizes related work, and Section 7 concludes.

2 System Architecture

This section introduces the *Virtual eXecutable Archive* (VXA) architecture at a high level. The principles described in this section are generic and should be applicable to data compression, backup, and archival storage systems of all kinds. All implementation details specific to the prototype VXA archiver and virtual machine are left for the next section.

2.1 Trends and Design Principles

Archived data is almost always compressed in some fashion to save space. The one-time cost of compressing the data in the first place is usually well justified by the savings in storage costs (and perhaps network bandwidth) offered by compression over the long term.

A basic property of data compression, however, is that the more you know about the data being compressed, the more effectively you can compress it. General string-oriented compressors such as `gzip` do not perform well on digitized photographs, audio, or video, because the information redundancy present in digital media does not predominantly take the form of repeated byte strings, but is specific to the type of media. For this reason a wide variety of media-specific compressors have appeared recently. *Lossless* compressors achieve moderate compression ratios while preserving all original information content, while *lossy* compressors achieve higher compression ratios by discarding information whose loss is deemed “unlikely to be missed” based on semantic knowledge of the data. Specialization of compression algorithms is not limited to digital media: compressors for semistructured data such as XML are also available for example [26]. This trend toward specialized encodings leads to a first important design principle for efficient archival storage:

An archival storage system must permit use of multiple, specialized compression algorithms.

Strong economic demand for ever more sophisticated and effective data compression has led to a rapid evolution in encoding schemes, even within particular domains such as audio or video, often yielding an abundance of mutually-incompatible competing schemes. Even when open standards achieve widespread use, the dominant standards evolve over time: e.g., from Unix `compress` to `gzip` to `bzip2`. This trend leads to VXA’s second basic design principle:

An archival storage system must permit its set of compression algorithms to evolve regularly.

The above two trends unfortunately work against the basic purpose of archival storage: to store data so that it remains available and usable later, perhaps decades later. Even if data is always archived using the latest encoding software, that software—and the operating systems it runs on—may be long obsolete a few years later when the archived data is needed. The widespread use of lossy encoding schemes compounds this problem, because periodically decoding and re-encoding archived data using the latest schemes would cause progressive information loss and thus is not generally a viable option. This constraint leads to VXA’s third basic design principle:

Archive extraction must be possible without specific knowledge of the data’s encoding.

VXA satisfies these constraints by storing executable decoders with all archived data, and by ensuring that these decoders run in a simple, well-defined, portable, and thus hopefully relatively “future-proof” virtual environment.

2.2 Creating Archives

Figure 3 illustrates the basic structure of an archive writer in the VXA architecture. The archiver contains a number of encoder/decoder or *codec* pairs: several specialized codecs designed to handle specific content types such as audio, video, or XML, and at least one general-purpose lossless codec. The archiver’s codec set is extensible via plug-ins, allowing the use of specialized codecs for domain-specific content when desired.

The archiver accepts both uncompressed and already-compressed files as inputs, and automatically tries to compress previously uncompressed input files using a scheme appropriate for the file’s type if available. The archiver attempts to compress files of unrecognized type using a general-purpose lossless codec such as `gzip`. By default the archiver uses only lossless encoding schemes for its automatic compression, but it may apply lossy encoding at the specific request of the operator.

The archiver writes into the archive a copy of the decoder portion of each codec it uses to compress data. The archiver of course needs to include only one copy of a given decoder in the archive, amortizing the storage cost of the decoder over all archived files of that type.

The archiver’s codecs can also recognize when an input file is *already* compressed in a supported format. In this case, the archiver just copies the pre-compressed data into the archive, since re-compressing already-compressed data is generally ineffective and particularly undesirable when lossy compression is involved. The archiver still includes a copy of the appropriate decoder in the archive,

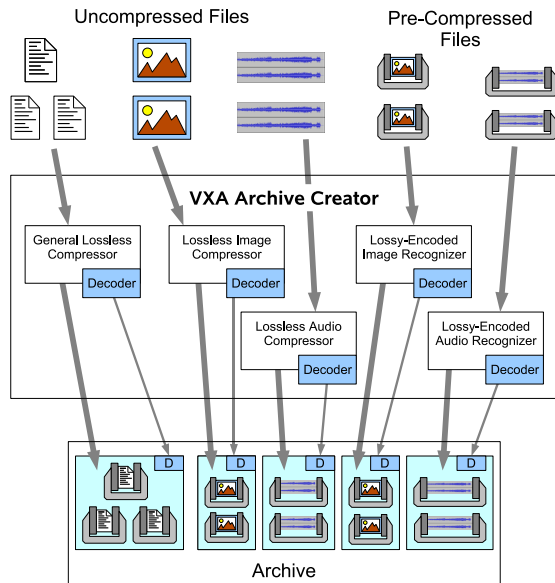


Figure 3: Archive Writer Operation

ensuring the data's continuing usability even after the original codec has become obsolete or unavailable.

Some of the archiver's codecs may be incapable of compression, but may instead merely recognize files already encoded using other, standalone compressors, and attach a suitable decoder to the archived file. We refer to such pseudo-codecs as *recognizer-decoders*, or *redecs*.

2.3 Reading Archives

Figure 4 illustrates the basic structure of the VXA archive reader. Unlike the writer, the reader does not require a collection of content-specific codecs, since all the decoders it needs are embedded in the archive itself. Instead, the archive reader implements a virtual machine in which to run those archived decoders. To decode a compressed file in the archive, the archive reader first locates the associated decoder in the archive and loads it into its virtual machine. The archive reader then executes the decoder in the virtual machine, supplying the encoded data to the decoder while accepting decoded data from the decoder, to produce the decompressed output file.

The archive reader by default only decompresses files that weren't already compressed when the archive was written. This way, archived files that were already compressed in popular standard formats such as JPEG or MP3, which tend to be widely and conveniently usable in their compressed form, remain compressed by default after extraction. The reader can, however, be forced to decode *all*

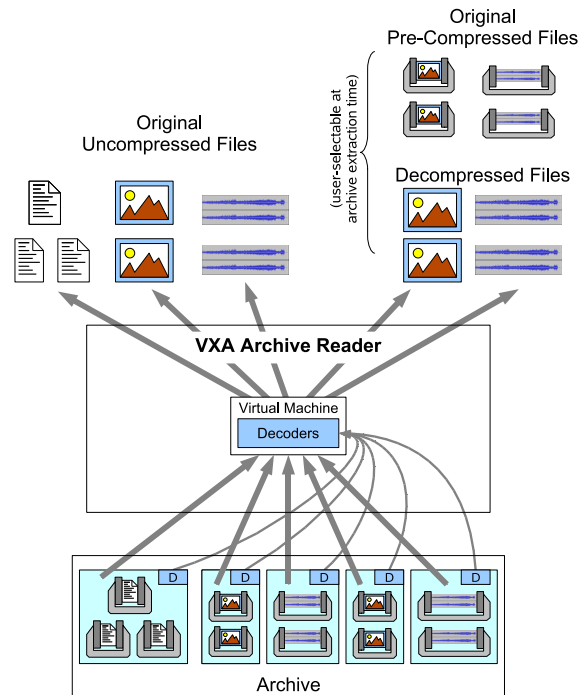


Figure 4: Archive Reader Operation

archived files having an associated decoder, as shown in Figure 4, ensuring that encoded data remains decipherable even if "native" decoders for the format disappear.

This capability also helps protect against data corruption caused by codec bugs or evolution of standards. If an archived audio file was generated by a buggy MP3 encoder, for example, it may not play properly later under a different MP3 decoder after extraction from the archive in compressed form. As long as the audio file was originally archived with the specific (buggy) MP3 decoder that can decode the file correctly, however, the archive reader can still be instructed to use that archived decoder to recover a usable decompressed audio stream.

The VXA archive reader does not *always* have to use the archived x86-based decoders whenever it extracts files from an archive. To maximize performance, the reader might by default recognize popular compressed file types and decode them using non-virtualized decoders compiled for the native host architecture. Such a reader would fall back on running a virtualized decoder from the archive when no suitable native decoder is available, when the native decoder does not work properly on a particular archived stream, or when explicitly checking the archive's integrity. Even if the archive reader commonly uses native rather than virtualized decoders, the presence of the VXA decoders in the archive provides a crucial long-term fall-

back path for decoding, ensuring that the archived information remains decipherable after the codec it was compressed with has become obsolete and difficult to find.

Routinely using native decoders to read archives instead of the archived VXA decoders, of course, creates the important risk that a bug in a VXA decoder might go unnoticed for a long time, making an archive seem work fine in the short term but be impossible to decode later after the native decoder disappears. For this reason, it is crucial that explicit archive integrity tests always run the archived VXA decoder, and in general it is safest if the archive reader always uses the VXA decoder even when native decoders are available. Since users are unlikely to adopt this safer operational model consistently unless VXA decoder efficiency is on par with native execution, the efficiency of decoder virtualization is more important in practice than it may appear in theory.

2.4 The VXA Virtual Machine

The archive reader's virtual machine isolates the decoders it runs from both the host operating system and the processor architecture on which the archive reader itself runs. Decoders running in the VXA virtual machine have access to the computational primitives of the underlying processor but are extremely limited in terms of input/output. The only I/O decoders are allowed is to read an encoded data stream supplied by the archive reader and produce a corresponding decoded output stream. Decoders cannot access any host operating system services, such as to open files, communicate over the network, or interact with the user.

Through this strong isolation, the virtual machine not only ensures that decoders remain generic and portable across many generations of operating systems, but it also protects the host system from buggy or malicious decoders that may be embedded in an archive. Assuming the virtual machine is implemented correctly, the worst harm a decoder can cause is to garble the data it was supposed to produce from a particular encoded file. Since a decoder cannot communicate, obtain information about the host system, or even check the current system time, decoders do not have access to information with which they might deliberately "sabotage" their data based on the conditions under which they are run.

When an archive contains many files associated with the same decoder, the archive reader has the option of re-initializing the virtual machine with a pristine copy of the decoder's executable image before processing each new file, or reusing the virtual machine's state to decode multiple files in succession. Reusing virtual machine state may improve performance, especially on archives containing

many small files, at the cost of introducing the risk that a buggy or malicious decoder might "leak" information from one file to another during archive extraction, such as from a sensitive password or private key file to a multimedia stream that is likely to appear on a web page. The archive reader can minimize this security risk in practice by always re-initializing the virtual machine whenever the security attributes of the files it is processing change, such as Unix owner/group identifiers and permissions.

The VXA virtual machine is based on the standard 32-bit x86 architecture: all archived decoder executables are represented as x86-32 code, regardless of the actual processor architecture of the host system. The choice of the ubiquitous x86-32 architecture ensures that almost any existing decoder written in any language can be easily ported to run on the VXA virtual machine.

Although continuous improvements in processor hardware are likely to make the performance of an archived VXA decoder largely irrelevant over the long term, compressed archives are frequently used for more short-term purposes as well, such as making and restoring backups, distributing and installing software, and packaging XML-based structured documents [43]. Archive extraction performance is crucial to these short-term uses, and an archival storage system that performs poorly now is unlikely to receive widespread adoption regardless of its long-term benefits. Besides supporting the re-use of existing decoder implementations, VXA's adoption of the x86 architecture also enables those decoders to run quite efficiently on x86-based host processors, as demonstrated later in Section 5. Implementing the VM efficiently on other architectures requires binary translation, which is more difficult and may be less efficient, but is nevertheless by now a practical and proven technology [40, 9, 14, 3].

2.5 Applicability

The VXA architecture does not address the complete problem of preserving the long-term usability of archived digital information. The focus of VXA is on preserving *compressed* data streams, for which simpler uncompressed formats are readily available that can represent the same information. VXA will not necessarily help with old proprietary word processor documents, for example, for which there is often no obvious "simpler form" that preserves all of the original semantic information.

Many document processing applications, however, are moving toward use of "self-describing" XML-based structured data formats [43], combined with a general-purpose "compression wrapper" such as ZIP [21] for storage efficiency. The VXA architecture may benefit the

compression wrapper in such formats, allowing applications to encode documents using proprietary or specialized algorithms for efficiency while preserving the interoperability benefits of XML. VXA's support for specialized compression schemes may be particularly important for XML, in fact, since "raw" XML is extremely space-inefficient but can be compressed most effectively given some specialized knowledge of the data [26].

3 Archiver Implementation

Although the basic VXA architecture as described above could be applied to many archival storage or backup systems, the prototype implementation explored in this paper takes the form of an enhancement to the venerable ZIP/UnZIP archival tools [21]. The ZIP format was chosen over the `tar/gzip` format popular on Unix systems because ZIP compresses files individually rather than as one continuous stream, making it amenable to treating files of different types using different encoders.

For clarity, we will refer to the new VXA-enhanced ZIP and UnZIP utilities here as `vxZIP` and `vxUnZIP`, and to the modified archive format as "vxZIP format." In practice, however, the new tools and archive format can be treated as merely a natural upgrade to the existing ones.

3.1 ZIP Archive Format Modifications

The enhanced `vxZIP` archive format retains the same basic structure and features as the existing ZIP format, and the new utilities remain backward compatible with archives created with existing ZIP tools. Older ZIP tools can list the contents of archives created with `vxZIP`, but cannot extract files requiring a VXA decoder.

The ZIP file format historically uses a relatively fixed, though gradually growing, collection of general-purpose lossless codecs, each identified by a "compression method" tag in a ZIP file. A particular ZIP utility generally compresses all files using only one algorithm by default—the most powerful algorithm it supports—and UnZIP utilities include built-in decoders for most of the compression schemes used by past ZIP utilities. (Decoders for the old LZW-based "shrinking" scheme were commonly omitted for many years due to the LZW patent [4], illustrating one of the practical challenges to preserving archived data usability.)

In the enhanced `vxZIP` format, an archive may contain files compressed using a mixture of traditional ZIP compression methods and new VXA-specific methods. Files archived using traditional methods are assigned the standard method tag, permitting even VXA-unaware UnZIP

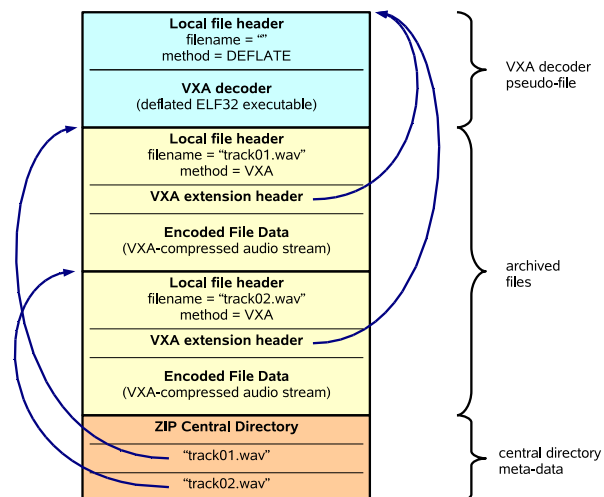


Figure 5: vxZIP Archive Structure

tools to identify and extract them successfully. The `vxZIP` format reserves one new "special" ZIP method tag for files compressed using VXA codecs that do not have their own ZIP method tags, and which thus can only be extracted with the help of an attached VXA decoder.

Regardless of whether an archived file uses a traditional or VXA compression scheme, `vxZIP` attaches a new VXA extension header to each file, pointing to the file's associated VXA decoder, as illustrated in Figure 5. Using this extension header, a VXA-aware archive reader can decode any archived file even if it has an unknown method tag. At the same time, `vxUnZIP` can still use a file's ZIP method tag to recognize files compressed using well-known algorithms for which it may have a faster native decoder.

When `vxZIP` recognizes an input file that is already compressed using a scheme for which it has a suitable VXA decoder, it stores the pre-compressed file directly without further compression and tags the file with compression method 0 (no compression). This method tag indicates to `vxUnZIP` that the file should normally be left compressed on extraction, and enables older UnZIP utilities to extract the file in its original compressed form. The `vxZIP` archiver nevertheless attaches a VXA decoder to the file in the same way as for automatically-compressed files, so that `vxUnZIP` can later be instructed to decode the file all the way to its uncompressed form if desired.

3.2 Archiving VXA Decoders

Since the 64KB size limitation of ZIP extension headers precludes storing VXA decoders themselves in the file headers, `vxZIP` instead stores each decoder elsewhere in

the archive as a separate “pseudo-file” having its own local file header and an empty filename. The VXA extension headers attached to “actual” archived files contain only the ZIP archive offset of the decoder pseudo-file. Many archive files can thus refer to one VXA decoder merely by referring to the same ZIP archive offset.

ZIP archivers write a *central directory* to the end of each archive, which summarizes the filenames and other meta-data of all files stored in the archive. The vxZIP archiver includes entries in the central directory only for “actual” archived files, and not for the pseudo-files containing archived VXA decoders. Since UnZIP tools normally use the central directory when listing the archive’s contents, VXA decoder pseudo-files do not show up in such listings even using older VXA-unaware UnZIP tools, and old tools can still use the central directory to find and extract any files not requiring VXA-specific decoders.

A VXA decoder itself is simply an ELF executable for the 32-bit x86 architecture [45], as detailed below in Section 4. VXA decoders are themselves compressed in the archive using a fixed, well-known algorithm: namely the ubiquitous “deflate” method used by existing ZIP tools and by the `gzip` utility popular on Unix systems.

3.3 Codecs for the Archiver

Since a basic goal of the VXA architecture is to be able to support a wide variety of often specialized codecs, it is unacceptable for vxZIP to have a fixed set of built-in compressors, as was generally the case for previous ZIP tools. Instead, vxZIP introduces a plug-in architecture for codecs to be used with the archiver. Each codec consists of two main components:

- The encoder is a standard dynamic-link library (DLL), which the archiver loads into its own address space at run-time, and invokes directly to recognize and compress files. The encoder thus runs “natively” on the host processor architecture and in the same operating system environment as the archiver itself.
- The decoder is an executable image for the VXA virtual machine, which the archiver writes into the archive if it produces or recognizes any encoded files using this codec. The decoder is always an ELF executable for the 32-bit x86 architecture implemented by the VXA virtual machine, regardless of the host processor architecture and operating system on which the archiver actually runs.

A natural future extension to this system would be to run VXA encoders as well as decoders in a virtual machine, making complete codec pairs maximally portable.

While such an extension should not be difficult, several tradeoffs are involved. A virtual machine for VXA encoders may require user interface support to allow users to configure encoding parameters, introducing additional system complexity. While the performance impact of the VXA virtual machine is not severe at least on x86 hosts, as demonstrated in Section 5, implementing encoders as native DLLs enables the archiving process to run with maximum performance on any host. Finally, vendors of proprietary codecs may not wish to release their encoders for use in a virtualized environment, because it might make license checking more difficult. For these reasons, virtualized VXA encoders are left for future work.

4 The Virtual Machine

The most vital component of the vxUnZIP archive reader is the virtual machine in which it runs archived decoders. This virtual machine is implemented by vx32, a novel *virtual machine monitor* (VMM) that runs in user mode as part of the archive reader’s process, without requiring any special privileges or extensions to the host operating system. Decoders under vx32 effectively run within vxUnZIP’s address space, but in a software-enforced fault isolation domain [46], protecting the application process from possible actions of buggy or malicious decoders. The VMM is implemented as a shared library linked into vxUnZIP; it can also be used to implement specialized virtual machines for other applications.

The vx32 VMM currently runs only on x86-based host processors, in both 32-bit and the new 64-bit modes. The VMM relies on quick x86-to-x86 code scanning and translation techniques to sandbox a decoder’s code as it executes. These techniques are comparable to those used by Embra [48], VMware [42], and Valgrind [34], though vx32 is simpler as it need only provide isolation, and not simulate a whole physical PC or instrument object code for debugging. Full binary translation to make vx32 run on other host architectures is under development.

4.1 Data Sandboxing

The VXA virtual machine provides decoders with a “flat” unsegmented address space up to 1GB in size, which always starts at virtual address 0 from the perspective of the decoder. The VM does not allow decoders access to the underlying x86 architecture’s legacy segmentation facilities. The vx32 VMM does, however, *use* the legacy segmentation features of the x86 host processor in order to implement the virtual machine efficiently.

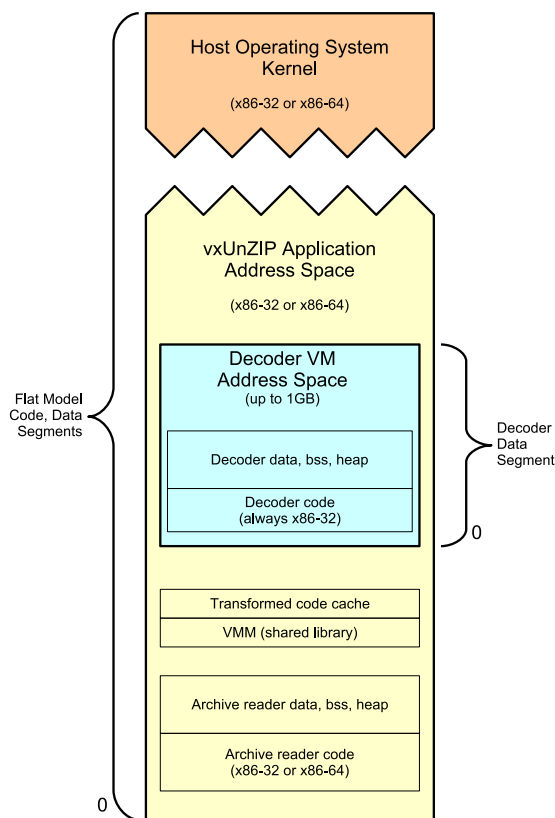


Figure 6: Archive Reader and VMM Address Spaces

As illustrated in Figure 6, vx32 maps a decoder’s virtual address space at some arbitrary location within its own process, and sets up a special process-local (LDT) data segment with a base and limit that provides access only to that region. While running decoder code, the VMM keeps this data segment loaded into the host processor’s segment registers that are used for normal data reads and writes (DS, ES, and SS). The decoder’s computation and memory access instructions are thus automatically restricted to the sandbox region, without requiring the special code transformations needed on other architectures [46].

Although the legacy segmentation features that the VMM depends on are not functional in the 64-bit addressing mode (“long mode”) of the new x86-64 processors, these processors provide 64-bit applications the ability to switch back to a 32-bit “compatibility mode” in which segmentation features are still available. On a 64-bit system, vxUnZIP and the VMM run in 64-bit long mode, but decoders run in 32-bit compatibility mode. Thus, vx32 runs equally well on both x86-32 and x86-64 hosts with only minor implementation differences in the VMM (amounting to about 100 lines of code).

4.2 Code Sandboxing

Although the VMM could similarly set up an x86 code segment that maps only the decoder’s address space, doing so would not by itself prevent decoders from executing arbitrary x86 instructions that are “unsafe” from the perspective of the VMM, such as those that would modify the segment registers or invoke host operating system calls directly. On RISC-based machines with fixed instruction sizes, a software fault isolation VMM can solve this problem by scanning the untrusted code for “unsafe” code sequences when the code is first loaded [46]. This solution is not an option on the x86’s variable-length instruction architecture, unfortunately, because within a byte sequence comprising one or more legitimate instructions there may be sub-sequences forming unsafe instructions, to which the decoder code might jump directly. The RISC-based techniques also reserve up to five general-purpose registers as *dedicated registers* to be used for fault isolation, which is not practical on x86-32 since the architecture provides only eight general-purpose registers total.

The vx32 VMM therefore never executes decoder code directly, but instead dynamically scans decoder code sequences to be executed and transforms them into “safe” code fragments stored elsewhere in the VMM’s process. As with Valgrind [34] and just-in-time compilation techniques [15, 24], the VMM keeps transformed code fragments in a cache to be reused whenever the decoder subsequently jumps to the same virtual entrypoint again.

The VMM must of course transform all flow control instructions in the decoder’s original code so as to keep execution confined to the safe, transformed code. The VMM rewrites branches with fixed targets to point to the correct transformed code fragment if one already exists. Branches to fixed but as-yet-unknown targets become branches to a “trampoline” that, when executed, transforms the target code and then back-patches the original (transformed) branch instruction to point directly to the new target fragment. Finally, the VMM rewrites indirect branches whose target addresses are known only at runtime (including function return instructions), so as to look up the target address dynamically in a hash table of transformed code entrypoints.

4.3 Virtual System Calls

The vx32 VMM rewrites x86 instructions that would normally invoke system calls to the host operating system, so as to return control to the user-mode VMM instead. In this way, vx32 ensures that decoders have no direct access to host OS services, but can only make controlled “virtual system calls” to the VMM or the archive reader.

Only five virtual system calls are available to decoders running under vxUnZIP: `read`, `write`, `exit`, `setperm`, and `done`. The first three have their standard Unix meanings, while `setperm` supports heap memory allocation, and `done` enables decoders to signal to vxUnZIP that they have finished decoding one stream and are able to process another without being re-loaded. Decoders have access to three standard “virtual file handles”—`stdin`, `stdout`, and `stderr`—but have no way to open any other files. A decoder’s virtual `stdin` file handle represents the data stream to be decoded, its `stdout` is the data stream it produces by decoding the input, and `stderr` serves the traditional purpose of allowing the decoder to write error or debugging messages. (vxUnZIP only displays such messages from decoders when in verbose mode.) A VXA decoder is therefore a traditional Unix filter in a very pure form.

Since a decoder’s address space comprises a portion of vxUnZIP’s own address space, the archive reader can easily access the decoder’s data directly for the purpose of servicing virtual system calls, in the same way that the host OS kernel services system calls made by application processes. To handle the decoder’s `read` and `write` calls, vxUnZIP merely passes the system call on to the native host OS after checking and adjusting the file handle and buffer pointer arguments. A decoder’s I/O calls thus require no extra data copying, and the indirection through the VMM and vxUnZIP code is cheap as it does not cross any hardware protection domains.

5 Evaluation and Results

This section experimentally evaluates the prototype vxZIP/vxUnZIP tools in order to analyze the practicality of the VXA architecture. The two most obvious questions about the practicality of VXA are whether running decoders in a virtual machine seriously compromises their performance for short-term uses of archives such as backups and software/data packaging, and whether embedding decoders in archives entails a significant storage cost. We also consider the portability issues of implementing virtual machines that run x86-32 code on other hosts.

5.1 Test Decoders

The prototype vxZIP archiver includes codecs for several well-known compressed file formats, summarized in Table 1. The two general-purpose codecs, `zlib` and `bzip2`, are for arbitrary data streams: vxZIP can use either of them as its “default compressor” to compress

files of unrecognized type while archiving. The remaining codecs are media-specific. All of the codecs are based directly on publicly-available libraries written in C, and were compiled using a basic GCC cross-compiler setup.

The `jpeg` and `jp2` codecs are recognizer-decoders (“redec”), which recognize still images compressed in the lossy JPEG [47] and JPEG-2000 [23] formats, respectively, and attach suitable VXA decoders to archived images. These decoders, when run under vxUnZIP, output uncompressed images in the simple and universally-understood Windows BMP file format. The `vorbis` redec similarly recognizes compressed audio streams in the lossy Ogg/Vorbis format [49], and attaches a Vorbis decoder that yields an uncompressed audio file in the ubiquitous Windows WAV audio file format.

Finally, `flac` is a full encoder/decoder pair for the Free Lossless Audio Codec (FLAC) format [11]. Using this codec, vxZIP can not only recognize audio streams already compressed in FLAC format and attach a VXA decoder, but it can also recognize *uncompressed* audio streams in WAV format and automatically compress them using the FLAC encoder. This codec thus demonstrates how a VXA archiver can make use of compression schemes specialized to particular types of data, without requiring the archive reader to contain built-in decoders for each such specialized compression scheme.

The above codecs with widely-available open source implementations were chosen for purposes of *evaluating* the prototype vxZIP/vxUnZIP implementation, and are not intended to serve as ideal examples to *motivate* the VXA architecture. While the open formats above may gradually evolve over time, their open-source decoder implementations are unlikely to disappear soon. Commercial archival and multimedia compression products usually incorporate proprietary codecs, however, which might serve as better “motivating examples” for VXA: proprietary codecs tend to evolve more quickly due to intense market pressures, and their closed-source implementations cannot be maintained by the customer or ported to new operating systems once the original product is obsolete and unsupported by the vendor.

5.2 Performance of Virtualized Decoders

To evaluate the performance cost of virtualization, the graph in Figure 7 shows the user-mode CPU time consumed running the above decoders over several test data sets, both natively and under the vx32 VMM. All execution times are normalized to the native execution time on an x86-32 host system. The data set used to test the general-purpose lossless codes is a Linux 2.6.11 ker-

Decoder	Description	Availability	Output Format
General-Purpose Codecs			
zlib	"Deflate" algorithm from ZIP/gzip	www.zlib.net	(raw data)
bzip2	Popular BWT-based algorithm	www.bzip.org	(raw data)
Still Image Codecs			
jpeg	Independent JPEG Group (IJG) reference decoder	www.iijg.org	BMP image
jp2	JPEG-2000 reference decoder from JasPer library	www.jpeg.org/jpeg2000	BMP image
Audio Codecs			
flac	Free Lossless Audio Codec (FLAC) decoder	flac.sourceforge.net	WAV audio
vorbis	Ogg Vorbis audio decoder	www.vorbis.com	WAV audio

Table 1: Decoders Implemented in vxZIP/vxUnZIP Prototype

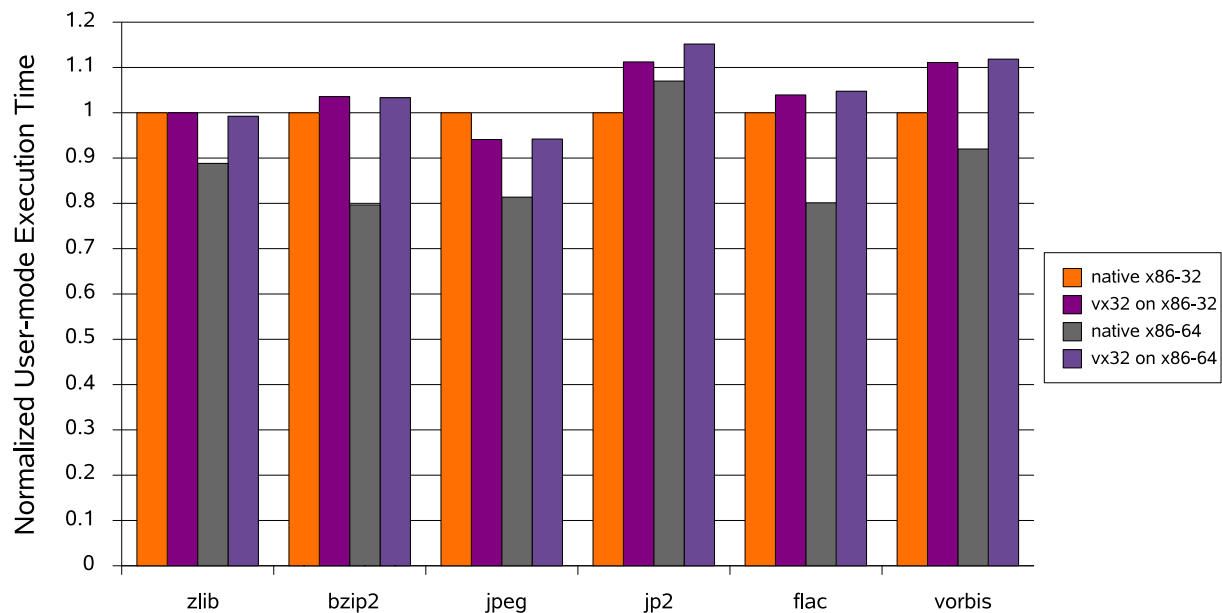


Figure 7: Performance of Virtualized Decoders

nel source tree; the data sets used for the media-specific codecs consist of typical pictures and music files in the appropriate format. All tests were run on an AMD Athlon 64 3000+ with 512MB of RAM, on both the x86-32 and x86-64 versions of SuSE Linux 9.3. The same compiler version (GCC 4.0.0) and optimization settings (`-O3`) were used for the native and virtualized versions of each decoder, and the timings represent user-mode process time as reported by the `time` command so as to factor out disk and system overhead. Total wall-clock measurements are not shown because for all but the slowest decoder, `jp2`, disk overhead dominates total wall-clock time and introduces enough additional variance between successive runs to swamp the differences in CPU-bound decoding time.

As Figure 7 shows, the decoders running under the vx32 VMM experience a slowdown of up to 11% relative to native x86-32 execution. The `vorbis` decoder initially experienced a 29% slowdown when compiled for VXA unmodified, due to subroutine calls in the decoder's inner loop that accentuate the VMM's flow-control overhead by requiring hash table lookups (see Section 4.2). Inlining these two functions both improved the performance of the native decoder slightly (about 1%) and reduced the relative cost of virtualization to 11%. The other decoders were unmodified from their original distribution form. The JPEG decoder became slightly faster under vx32, possibly due to effects of the VMM's code rewriting on instruction cache locality; such effects are possible and have been exploited elsewhere [2].

The virtualized decoders fall farther behind in comparison with native execution on an x86-64 host, but this difference is mostly due to the greater efficiency of the 64-bit native code rather than to virtualization overhead. Virtualized decoders always run in 32-bit mode regardless of the host system, so their absolute performance is almost identical on 32-bit versus 64-bit hosts, as the graph shows.

5.3 Decoder Storage Overhead

To evaluate the storage overhead of embedding decoders in archives, Table 2 summarizes the size of each decoder's executable image when compiled and linked for the VXA virtual machine. The code size for each decoder is further split into the portion comprising the decoder itself versus the portion derived from the statically-linked C library against which each decoder is linked. No special effort was made to trim unnecessary code, and the decoders were compiled to optimize performance over code size.

The significance of these absolute storage overheads of course depends on the size of the archive in which they are embedded, since only one copy of a decoder needs to be stored in the archive regardless of the number of encoded files that use it. As a comparison point, however, a single 2.5-minute CD-quality song in the dataset used for the earlier performance tests, compressed at 120Kbps using the lossy Ogg codec, occupies 2.2MB. The 130KB Ogg decoder for VXA therefore represents a 6% space overhead in an archive containing only this one song, or a 0.6% overhead in an archive containing a 10-song album. The same 2.5-minute song compressed using the lossless FLAC codec occupies 24MB, next to which the 48KB vx32 decoder represents a negligible 0.2% overhead.

5.4 Portability Considerations

A clear disadvantage of using the native x86 processor architecture as the basis for VXA decoders is that porting the archive reader to non-x86 host architectures requires instruction set emulation or binary translation. While instruction set emulators can be quite portable, they also tend to be many times slower than native execution, making them unappealing for computation-intensive tasks such as data compression. Binary translation provides better performance and has entered widespread commercial use, but is not simple to implement, and even the best binary translators are unlikely to match the performance of natively-compiled code.

The QEMU x86 emulator [6] introduces a binary translation technique that offers a promising compromise between portability and performance. QEMU uses a native C compiler for the host processor architecture to generate

short code fragments that emulate individual x86 instructions. QEMU's dynamic translator then scans the x86 code at run-time and pastes together the appropriate native code fragments to form translated code. While this method is unlikely to perform as well as a binary translator designed and optimized for a specific host architecture, it provides a portable method of implementing emulators that offer usable performance levels.

Even without efficient binary translation for x86 code, however, the cost of emulation does not necessarily make the VXA architecture impractical for non-x86 host architectures. An archive reader can still provide fast native decoders for currently popular file formats, running archived decoders under emulation only when no native decoder is available. The resulting archival system is no slower in practice than existing tools based on a fixed set of compressors, but provides the added assurance that archived data will still be decipherable far into the future. It is much better to be able to decode archived data slowly using emulation than not to be able to decode it at all.

5.5 Availability

The vxZIP/vxUnZIP tools, the vx32 virtual machine, and the data sets used in the above tests can be obtained from <http://pdos.csail.mit.edu/~baford/vxa/>.

6 Related Work

The importance and difficulty of preserving digital information over the long term is gaining increasing recognition [16]. This problem can be broken into two components: preserving *data* and preserving the data's *meaning* [13]. Important work is ongoing to address the first aspect [17, 12, 30], and the second, the focus of this paper, is beginning to receive serious attention.

6.1 Archival Storage Strategies

Storing executable decoders with archived data is not new: popular archivers including ZIP often ship with tools to create *self-extracting archives*, or executables that decompress themselves when run [35, 21]. Such self-extracting archives are designed for convenience, however, and are traditionally specific to a particular host operating system, making them as bad as or worse than traditional non-executable archives for data portability and longevity. Self-extracting archives also provide no security against bugs or malicious decoders; E-mail viruses routinely disguise themselves as self-extracting archives supposedly containing useful applications.

Decoder	Code Size					Compressed (zlib)
	Total	Decoder		C Library		
General-Purpose Codecs						
zlib	46.0KB	32.4KB	(70%)	13.6KB	(30%)	26.2KB
bzip2	71.1KB	60.9KB	(86%)	10.2KB	(14%)	29.9KB
Still Image Codecs						
jpeg	103.3KB	90.0KB	(87%)	13.3KB	(13%)	48.6KB
jp2	220.2KB	198.5KB	(90%)	21.7KB	(10%)	105.9KB
Audio Codecs						
flac	102.5KB	84.2KB	(82%)	18.3KB	(18%)	47.6KB
vorbis	233.4KB	200.3KB	(86%)	33.1KB	(14%)	129.7KB

Table 2: Code Size of Virtualized Decoders

Rothenberg suggested a decade ago the idea of archiving the original application and system software used to create data along with the data itself, and using emulators to run archived software after its original hardware platform becomes obsolete [38]. Archiving entire systems and emulating their hardware accurately is difficult, however, because real hardware platforms (including necessary I/O devices) are extremely complex and tend to be only partly standardized and documented [5]. Preserving the *functionality* of the original system is also not necessarily equivalent to preserving the *usefulness* of the original data. The ability to view old data in an emulator window via the original application’s archaic user interface, for example, is not the same as being able to load or “cut-and-paste” the data into new applications or process it using new indexing or analysis tools.

Lorie later proposed to archive data along with specialized decoder programs, which run on a specialized “Universal Virtual Computer” (UVC), and extract archived data into a self-describing XML-like format [27]. The UVC’s simplicity makes emulation easier, but since it represents a new architecture substantially different from those of real processors, UVC decoders must effectively be written from scratch in assembly language until high-level languages and tools are developed [28]. More importantly, the UVC’s specialization to the “niche” of long-term archival storage systems virtually guarantees that high-level languages, development tools, and libraries for it will never be widely available or well-supported as they are for general-purpose architectures.

The LOCKSS archival system supports data format converter plug-ins that transparently migrate data in obsolete formats to new formats when a user accesses the data [37]. Over time, however, actively maintaining converter plug-ins for an ever-growing array of obsolete compressed formats may become difficult. Archiving VXA

decoders with compressed data *now* ensures that future LOCKSS-style “migrate-on-access” converters will only need to read common historical *uncompressed* formats, such as BMP images or WAV audio files, and not the far more numerous and rapidly-evolving compressed formats. VXA therefore complements a “migrate-on-access” facility by reducing the number and variety of source formats the access-time converters must support.

6.2 Specialized Virtual Environments

Virtual machines and languages have been designed for many specialized purposes, such as printing [1], boot loading [20], Web programming [19, 29], packet filters [32] and other OS extensions [41], active networks [44], active disks [36], and grid computing [8]. In this tradition, VXA could be appropriately described as an architecture for “active archives.”

Similarly, dynamic code scanning and translation is widely used for purposes such as migrating legacy applications across processor architectures [40, 9, 3], simulating complete hardware platforms [48], run-time code optimization [2], implementing new processors [14], and debugging compiled code [34, 39]. In contrast with the common “retroactive” uses of virtual machines and dynamic translation to “rescue old code” that no longer runs on the latest systems, however, VXA applies these technologies *proactively* to preserve the long-term usability and portability of archived data, *before* the code that knows how to decompress it becomes obsolete.

Most virtual machines designed to support safe application extensions rely on type-safe languages such as Java [7]. In this case, the constraints imposed by the language make the virtual machine more easily portable across processor architectures, at the cost of requiring all untrusted code to be written in such a language. While

just-in-time compilation [15, 24] has matured to a point where type-safe languages perform adequately for most purposes, some software domains in which performance is traditionally perceived as paramount—such as data compression—remain resolutely attached to unsafe languages such as C and assembly language. Advanced digital media codecs also frequently take advantage of the SIMD extensions of modern processors [22], which tend to be unavailable in type-safe languages. The desire to support the many widespread open and proprietary data encoding algorithms whose implementations are only available in unsafe languages, therefore, makes type-safe language technology infeasible for the VXA architecture.

6.3 Isolation Technologies

The prototype vx32 VMM demonstrates a simple and practical software fault isolation (SFI) strategy on the x86, which achieves performance comparable to previous techniques designed for on RISC architectures [46], despite the fact that the RISC-based techniques are not easily applicable to the x86 as discussed in Section 4.2. RISC-based SFI, observed to incur a 15–20% overhead for full virtualization, can be trimmed to 4% overhead by sandboxing memory writes but not reads, thereby protecting the host application from active interference by untrusted code but not from snooping. Unfortunately, this weaker security model is probably not adequate for VXA: a functional but malicious decoder for multimedia files likely to be posted on the Web, for example, could scan the archive reader’s address space for data left over from restoring sensitive files such as passwords and private keys from a backup archive, and surreptitiously leak that information into the (public) multimedia output stream it produces.

The Janus security system [18] runs untrusted “helper” applications in separate processes, using hardware-based protection in conjunction with Solaris’s sophisticated process tracing facilities to control the supervised applications’ access to host OS system calls. This approach is more portable across processor architectures than vx32’s, but less portable across operating systems since it relies on features currently unique to Solaris. The Janus approach also does not enhance the portability of the helper applications, since it does not insulate them from those host OS services they *are* allowed to access.

The L4 microkernel used an x86-specific segmentation trick analogous to vx32’s data sandboxing technique to implement fast IPC between small address spaces [25]. A Linux kernel extension similarly used segmentation and paging in combination to give user-level applications a sandbox for untrusted extensions [10]. This latter tech-

nique can provide each application with only one virtual sandbox at a time, however, and it imposes constraints on the kernel’s own use of x86 segments that would make it impossible to grant use of this facility to 64-bit applications on new x86-64 hosts.

7 Conclusion

The VXA architecture for archival data storage offers a new and practical solution to the problem of preserving the usability of digital content. By including executable decoders in archives that run on a simple and OS-independent virtual machine based on the historically enduring x86 architecture, the VXA architecture ensures that archived data can always be decoded into simpler and less rapidly-evolving uncompressed formats, long after the original codec has become obsolete and difficult to find. The prototype vxZIP/vxUnZIP archiver for x86-based hosts is portable across operating systems, and decoders retain good performance when virtualized.

Acknowledgments

The author wishes to thank Frans Kaashoek, Russ Cox, Maxwell Krohn, and the anonymous reviewers for many helpful comments and suggestions.

References

- [1] Adobe Systems Inc. *PostScript Language Reference*. Addison Wesley, 3rd edition, March 1999.
- [2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
- [3] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *36th International Conference on Microarchitecture (MICRO36)*, San Diego, CA, December 2003.
- [4] Michael C. Battilana. The GIF controversy: A software developer’s perspective, June 2004. <http://lzw.info/>.
- [5] David Bearman. Reality and chimeras in the preservation of electronic records. *D-Lib Magazine*, 5(4), April 1999.
- [6] Fabrice Bellard. QEMU, a fast and portable dynamic translator, April 2005.
- [7] Brian Case. Implementing the Java virtual machine. *Microprocessor Report*, 10(4):12–17, March 1996.
- [8] B. Chang, K. Crary, M. DeLap, R. Harper, J. Liska, T. Murphy VII, and F. Pfenning. Trustless grid computing in ConCert. In *Workshop on Grid Computing*, pages 112–125, Baltimore, MD, November 2002.

- [9] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates. FX!32: a profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March 1998.
- [10] Tzi-cker Chiueh, Ganesh Venkitachalam, and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Symposium on Operating System Principles*, pages 140–153, December 1999.
- [11] Josh Coalson. Free lossless audio codec format. <http://flac.sourceforge.net/format.html>.
- [12] Brian Cooper and Hector Garcia-Molina. Peer-to-peer data trading to preserve information. *Information Systems*, 20(2):133–170, 2002.
- [13] Arturo Crespo and Hector Garcia-Molina. Archival storage for digital libraries. In *International Conference on Digital Libraries*, 1998.
- [14] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *International Symposium on Code Generation and Optimization*, pages 15–24. IEEE Computer Society, 2003.
- [15] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, pages 297–302, Salt Lake City, UT, January 1984.
- [16] John Garrett and Donald Waters. Preserving digital information: Report of the task force on archiving of digital information, May 1996.
- [17] Andrew V. Goldberg and Peter N. Yianilos. Towards an archival intermemory. In *IEEE Advances in Digital Libraries*, pages 147–156, Santa Barbara, CA, 1998. IEEE Computer Society.
- [18] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *6th USENIX Security Symposium*, San Jose, CA, 1996.
- [19] James Gosling and Henry McGilton. The Java language environment, May 1996. <http://java.sun.com/docs/white/langenv/>.
- [20] IEEE. Std 1275-1994: Boot firmware, 1994.
- [21] Info-ZIP. <http://www.info-zip.org/>.
- [22] Intel Corporation. IA-32 Intel architecture software developer’s manual, June 2005.
- [23] International Standards Organization. JPEG 2000 image coding system, 2000. ISO/IEC 15444-1.
- [24] Andreas Krall. Efficient JavaVM just-in-time compilation. In *Parallel Architectures and Compilation Techniques*, pages 54–61, Paris, France, October 1998.
- [25] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Technical Report No. 933, GMD — German National Research Center for Information Technology, November 1995.
- [26] Hartmut Liefke and Dan Suciu. XMill: an efficient compressor for XML data. Technical Report MS-CIS-99-26, University of Pennsylvania, 1999.
- [27] Raymond A. Lorie. Long-term archiving of digital information. Technical Report RJ 10185, IBM Almaden Research Center, May 2000.
- [28] Raymond A. Lorie. The UVC: a method for preserving digital documents, proof of concept, 2002. IBM/KB Long-Term Preservation Study Report Series Number 4.
- [29] S. Lucco, O. Sharp, and R. Wahbe. Omniware: A Universal Substrate for Web Programming. *World Wide Web Journal*, 1(1):359–368, 1995.
- [30] Petros Maniatis, Mema Roussopoulos, T. J. Giuli, David S. H. Rosenthal, and Mary Baker. The LOCKSS peer-to-peer digital preservation system. *Transactions on Computing Systems*, 23(1):2–50, 2005.
- [31] Microsoft Corporation. Buffer overrun in JPEG processing (GDI+) could allow code execution (833987), September 2004. Microsoft Security Bulletin MS04-028.
- [32] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Symposium on Operating System Principles*, pages 39–51, Austin, TX, November 1987.
- [33] Mark Nelson. LZW data compression. *Dr. Dobbs’s Journal*, October 1989.
- [34] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. In *Third Workshop on Runtime Verification (RV’03)*, Boulder, CO, July 2003.
- [35] PKWARE Inc. PKZIP. <http://www.pkware.com/>.
- [36] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Very Large Databases (VLDB)*, New York, NY, August 1998.
- [37] David S. H. Rosenthal, Thomas Lipkis, Thomas S. Robertson, and Seth Morabito. Transparent format migration of preserved web content. *D-Lib Magazine*, 11(1), January 2005.
- [38] Jeff Rothenberg. Ensuring the longevity of digital documents. *Scientific American*, 272(1):24–29, January 1995.
- [39] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.
- [40] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, 1993.
- [41] Christopher Small and Margo Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, San Diego, CA, January 1996.
- [42] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation’s hosted virtual machine monitor. In *USENIX Annual Technical Conference*, Boston, MA, June 2001.
- [43] Sun Microsystems. OpenOffice.org XML file format 1.0, December 2002. xml.openoffice.org.
- [44] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, 1997.
- [45] Tool Interface Standard (TIS) Committee. Executable and linking format (ELF) specification, May 1995.
- [46] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.
- [47] G.K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, April 1991.
- [48] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.
- [49] Xiph.org Foundation. Vorbis I specification. <http://www.xiph.org/ogg/vorbis/>.

I/O System Performance Debugging Using Model-driven Anomaly Characterization*

Kai Shen Ming Zhong Chuanpeng Li

Department of Computer Science, University of Rochester

{kshen, zhong, cli}@cs.rochester.edu

Abstract

It is challenging to identify performance problems and pinpoint their root causes in complex systems, especially when the system supports wide ranges of workloads and when performance problems only materialize under particular workload conditions. This paper proposes a model-driven anomaly characterization approach and uses it to discover operating system performance bugs when supporting disk I/O-intensive online servers. We construct a whole-system I/O throughput model as the reference of expected performance and we use statistical clustering and characterization of performance anomalies to guide debugging. Unlike previous performance debugging methods offering detailed statistics at specific execution settings, our approach focuses on comprehensive anomaly characterization over wide ranges of workload conditions and system configurations.

Our approach helps us quickly identify four performance bugs in the I/O system of the recent Linux 2.6.10 kernel (one in the file system prefetching, two in the anticipatory I/O scheduler, and one in the elevator I/O scheduler). Our experiments with two Web server benchmarks, a trace-driven index searching server, and the TPC-C database benchmark show that the corrected kernel improves system throughput by up to five-fold compared with the original kernel (averaging 6%, 32%, 39%, and 16% for the four server workloads).

1 Introduction

It is not uncommon for complex systems to perform worse than expected. In the context of this paper, we define performance bugs as problems in system implementation that degrade the performance (compared with that intended by the design protocol/algorithm). Examples of such bugs include overly-simplified implementations, mis-management of special cases, or plain erroneous coding. These bugs, upon discovery, are typically quite easy to fix in comparison with implementing newer and better protocol/algorithms. However, it is challeng-

ing to identify performance problems and pinpoint their root causes in large software systems.

Previous techniques such as program instrumentation [13, 20], complete system simulation [24], performance assertion checking [22], and detailed overhead categorization [9] were proposed to understand performance problems in complex computer systems and applications. Some recent performance debugging work employs statistical analysis of online system traces [1, 7] to identify faulty components in large systems. In general, these techniques focus on offering fine-grained examination of the target system/application in specific execution settings. However, many systems (such as the I/O system in OS) are designed to support wide ranges of workload conditions and they may also be configured in various different ways. It is desirable to explore performance anomalies over a comprehensive universe of execution settings for these systems. Such exploration is particularly useful for performance debugging without the knowledge of runtime workload conditions and system configurations.

We propose a new approach that systematically characterizes performance anomalies in a system to aid performance debugging. The key advantage is that we can comprehensively consider wide ranges of workload conditions and system configurations. Our approach proceeds in the following steps (shown in Figure 1).

1. We construct a whole-system performance model according to the design protocol/algorithms of relevant system components. The model predicts system performance under different workload conditions and system configurations.
2. We acquire a representative set of anomalous workload and system configuration settings by comparing measured system performance with model prediction under a number of sample settings. For each system component that is considered for debugging, we include some sample settings where the component is bypassed.
3. We statistically cluster anomalous settings into groups likely attributed to individual “causes”. We then characterize each such cause (or bug) with correlated system component and workload conditions.

*This work was supported in part by NSF grants CCR-0306473, ITR/IIS-0312925, and an NSF CAREER Award CCF-0448413.

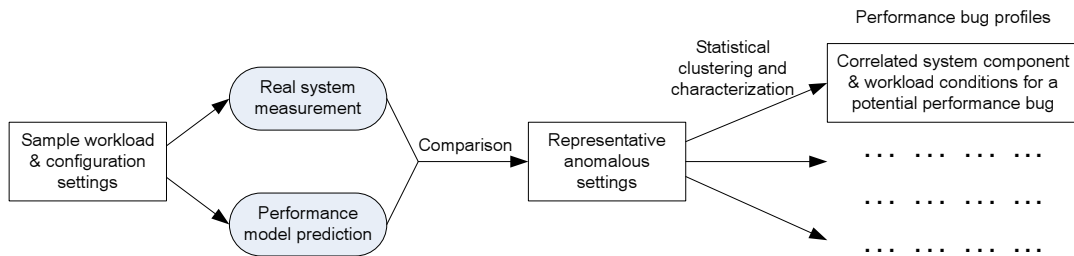


Figure 1: High-level overview of the proposed model-driven anomaly characterization.

The result of our approach contains profiles for potential performance bugs, each with a system component where the bug is likely located and the settings (workload conditions and system configurations) where it would inflict significant performance losses. Such result then assists further human debugging. It also helps verifying or explaining bugs after they are discovered. Even if some bugs could not be immediately fixed, our anomaly characterization identifies workload conditions and system configurations that should be avoided if possible.

Note that discrepancies between measured system performance and model prediction can also be caused by errors in the performance model. Therefore, we must examine both the performance model and the system implementation when presented with a bug profile. Since the performance model is much less complex in nature, we focus on debugging the system implementation in this paper.

It is possible for our approach to have false positives (producing characterizations that do not correspond to any real bugs) and false negatives (missing some bugs in the output). As a debugging aid where human screening is available, false positives are less of a concern. In order to achieve low false negatives, we sample wide ranges of workload parameters and various system configurations in a systematic fashion.

The rest of this paper presents our approach in details and describes our experience of discovering operating system performance bugs when supporting disk I/O-intensive online servers. Although our results in this paper focus on one target system and one type of workloads, we believe that the proposed model-driven anomaly characterization approach is general. It may assist the performance debugging of other systems and workloads as long as comprehensive performance models can be built for them.

2 Background

2.1 Targeted Workloads

The targeted workloads in this work are data-intensive online servers that access large disk-resident datasets

while serving multiple clients simultaneously. Examples include Web servers hosting large datasets and keyword search engines that support interactive search on terabytes of indexed Web pages. In these servers, each incoming request is serviced by a request handler which can be a thread in a multi-threaded server or a series of event handlers in an event-driven server. The request handler repeatedly accesses disk data and consumes CPU before completion. A request handler may block if the needed resource is unavailable. While request handlers consume both disk I/O and CPU resources, the overall server throughput is often dominated by I/O system performance when application data size far exceeds available server memory. For the ease of model construction in the next section, we assume that request handlers perform mostly read-only I/O when accessing disk-resident data. Many online services, such as Web server and index searching, do not involve any updates on hosted datasets.

Characteristics of the application workload may affect the performance of a disk I/O-intensive online server. For example, the data access locality and sequentiality largely determine how much of the disk time is spent on data transfer or seek and rotation.

2.2 Relevant Operating System Components

We describe operating system features that affect the I/O performance of data-intensive online servers.

Prefetching. Data accesses belonging to a single request handler often exhibit strong locality due to semantic proximity. During concurrent execution, however, data access of one request handler can be frequently interrupted by other active request handlers in the server. This may severely affect I/O efficiency due to long disk seek and rotational delays. The employment of OS prefetching can partially alleviate this problem. A larger prefetching depth increases the granularity of I/O requests, and consequently yields less frequent disk seeks and rotations. On the other hand, kernel-level prefetching may retrieve unneeded data due to the lack of knowledge on how much data is desired by the application. Such a waste tends to be magnified by aggressive prefetching policies.

I/O scheduling. Traditional elevator-style I/O schedulers such as Cyclic-SCAN sort and merge outstanding I/O requests to reduce the seek distance on storage devices. In addition, the anticipatory I/O scheduling [14] can be particularly effective for concurrent I/O workloads. At the completion of an I/O request, the anticipatory disk scheduler may choose to keep the disk idle for a short period of time even when there are pending requests. The scheduler does so in anticipation of a new I/O request from the same process that issued the just completed request, which often requires little or no seeking from the current disk head location. However, anticipatory scheduling may not be effective when substantial think time exists between consecutive I/O requests. The anticipation may also be rendered ineffective when a request handler has to perform interleaving synchronous I/O that does not exhibit strong locality. Such a situation arises when a request handler simultaneously accesses multiple data streams.

Others. For data-intensive workloads, memory caching is effective in improving the application-perceived performance over the raw storage I/O throughput. Most operating systems employ LRU-style policies to manage data cached in memory.

File system implementation issues such as file layout can also affect the system performance. We assume the file data is laid out contiguously on the storage. This is a reasonable assumption since the OS often tries to allocate file data contiguously on creation and the dataset is unchanged under our targeted read-only workloads.

3 I/O Throughput Model

Our model-driven performance debugging requires model-based prediction of the overall system performance under wide ranges of workload conditions and various system configurations. Previous studies have recognized the importance of constructing I/O system performance models. Various analytical and simulation models have been constructed for disk drives [5, 16, 25, 28, 36], disk arrays [8, 33], OS prefetching [6, 29, 31], and memory caching [15]. However, performance models for individual system components do not capture the inter-dependence of different components and consequently they may not accurately predict the overall application performance.

When modeling a complex system like ours, we follow the methodology of decomposing it into weakly coupled subcomponents. More specifically, we divide our whole-system I/O throughput model into four layers — OS caching, prefetching, OS-level I/O scheduling, and the storage device. Every layer may transform its input workload to a new workload imposed on the lower layer. For example, I/O scheduling may alter inter-request I/O

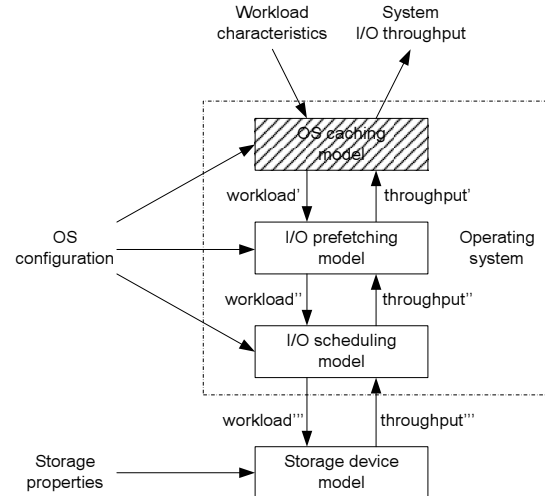


Figure 2: Layered system model on I/O throughput. We bypass the OS caching model in the context of this paper.

seek distances. Each layer may also change the predicted I/O throughput from the lower layer due to additional benefits or costs it may induce. For instance, prefetching adds the potential overhead of fetching unneeded data. As indicated in Figure 2, we use *workload*, *workload'*, *workload''*, and *workload'''* to denote the original and transformed workloads at each layer. We similarly use *throughput*, *throughput'*, *throughput''*, and *throughput'''* to represent the I/O throughput results seen at each layer.

Figure 2 illustrates our layered system model on I/O throughput. This paper focuses on the I/O system performance debugging and we bypass the OS caching model in our study. For the purpose of comparing our performance model with real system measurement, we add additional code in the operating system to disable the caching. More information on this is provided in Section 4.1. The rest of this section illustrates the other three layers of the I/O throughput model in detail. While mostly applicable to many general-purpose OSes, our model more closely follows the target system of our debugging work — the Linux 2.6 kernel.

3.1 OS Prefetching Model

We define a *sequential access stream* as a group of spatially contiguous data items that are accessed by a single request handler. Note that the request handler may not continuously access the entire stream at once. In other words, it may perform interleaving I/O that does not belong to the same stream. We further define a *sequential access run* as a portion of a sequential access stream that does not have such interleaving I/O. Figure 3 illustrates these two concepts. All read accesses from request handlers are assumed to be synchronous.

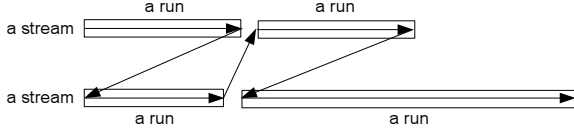


Figure 3: Illustration of the sequential access stream and the sequential access run. The arrows indicate the data access sequence of the request handler.

We consider the workload transformation of I/O prefetching on a sequential access stream of length S_{stream} . I/O prefetching groups data accesses of the stream into requests of size $S_{prefetch}$ — the I/O prefetching depth. Therefore, the number of I/O requests for serving this sequential stream is:

$$N_{request} = \lceil \frac{S_{stream}}{S_{prefetch}} \rceil \quad (1)$$

Operating system prefetching may retrieve unneeded data due to the lack of knowledge on how much data is desired by the application. In the transformed workload, the total amount of fetched data for the stream is:

$$S'_{stream} = \lceil \frac{S_{stream}}{S_{prefetch}} \rceil \cdot S_{prefetch} \quad (2)$$

Within the amount of fetched data S'_{stream} , the effective amount is only S_{stream} while the rest is not needed by the application. Therefore:

$$throughput' = throughput'' \cdot \frac{\sum S_{stream}}{\sum S'_{stream}} \quad (3)$$

However, wasted prefetching does not exist when each sequential access stream references a whole file since the OS would not prefetch beyond the end of a file. In this case, I/O prefetching does not fetch unneeded data and it does not change the I/O throughput. Therefore:

$$S'_{stream} = S_{stream} \quad (4)$$

$$throughput' = throughput'' \quad (5)$$

3.2 OS-level I/O Scheduling Model

The I/O scheduling layer passes the retrieved data to the upper layer without any change. Therefore it does not change the I/O throughput:

$$throughput'' = throughput''' \quad (6)$$

I/O scheduling transforms the workload primarily by sorting and merging I/O requests to reduce the seek distance on storage devices. We discuss such workload transformation by the traditional elevator-style I/O scheduling and by the anticipatory I/O scheduling.

3.2.1 Elevator-style I/O Scheduling

I/O scheduling algorithms such as Cyclic-SCAN reorder outstanding I/O requests based on data location and schedule the I/O request close to the current disk head location. The effectiveness of such scheduling is affected by the concurrency of the online server. Specifically, a smaller average seek distance can be attained at higher server concurrency when the disk scheduler can choose from more concurrent requests for seek reduction. We estimate that the number of simultaneous disk seek requests in the SCAN queue is equal to the server concurrency level γ . When the disk scheduler can choose from γ requests at uniformly random disk locations, a previous study [27] indicates that the inter-request seek distance D_{seek} follows the following distribution:

$$Pr[D_{seek} \geq x] = (1 - \frac{x}{\delta \cdot D_{disk}})^\gamma \quad (7)$$

Here δ is the proportion of the disk where the dataset resides and D_{disk} is the total disk size. In other words, $\delta \cdot D_{disk}$ represents the span of the dataset on the disk.

During concurrent execution (concurrency greater than one), the I/O scheduler switches to a different stream when a prefetching request from one stream is completed. Therefore it does not change the granularity of I/O requests passed from the prefetching layer. Consequently the average size of an I/O request is:

$$E(S_{request}) = \frac{\sum S'_{stream}}{\sum N_{request}} = \frac{E(S'_{stream})}{E(N_{request})} \quad (8)$$

At the concurrency of one, all I/O requests belonging to one sequential access run is merged:

$$E(S_{request}) = \max\{\frac{E(S'_{stream})}{E(N_{request})}, E(S_{run})\} \quad (9)$$

where $E(S_{run})$ is the average length of a sequential access run.

3.2.2 Anticipatory I/O Scheduling

During concurrent execution, the anticipatory I/O scheduling [14] may temporarily idle the disk so that consecutive I/O requests that belong to the same request handler are serviced without interruption. This effectively merges all prefetching requests of each sequential access run (defined in Section 3.1) into a single I/O request. Thus the average size of an I/O request in the transformed workload is:

$$E(S_{request}) = \max\{\frac{E(S'_{stream})}{E(N_{request})}, E(S_{run})\} \quad (10)$$

The anticipatory I/O scheduling likely reduces the frequency of disk seeks, but it does not affect the inter-request seek distance modeled in Equation (7).

The other effect of the anticipatory I/O scheduling is that it induces disk idle time during anticipatory waiting when useful work could be otherwise performed. The disk idle time for each I/O request T_{idle} is the total inter-request thinktime for the corresponding sequential access run.

3.3 Storage Device Model

Let the disk transfer rate be R_{tr} . Also let the seek time and rotational delay be T_{seek} and $T_{rotation}$ respectively. The disk resource consumption (in time) for processing a request of length $S_{request}$ includes a single seek, rotation, and the data transfer as well as the idle time:

$$T_{disk} = \frac{S_{request}}{R_{tr}} + T_{seek} + T_{rotation} + T_{idle} \quad (11)$$

Since $S_{request}$ is independent of R_{tr} , we have:

$$E(T_{disk}) = \frac{E(S_{request})}{E(R_{tr})} + E(T_{seek}) + E(T_{rotation}) + E(T_{idle}) \quad (12)$$

Therefore:

$$\begin{aligned} throughput''' &= \frac{E(S_{request})}{E(T_{disk})} \\ &= \frac{E(S_{request})}{\frac{E(S_{request})}{E(R_{tr})} + E(T_{seek}) + E(T_{rotation}) + E(T_{idle})} \end{aligned} \quad (13)$$

Below we determine the average data transfer rate $E(R_{tr})$, the average rotation delay $E(T_{rotation})$, and the average seek time $E(T_{seek})$. The sequential transfer rate depends on the data location (due to zoning on modern disks). With the knowledge of the data span on the disk and the histogram of data transfer rate at each disk location, we can then determine the average data transfer rate. We consider the average rotational delay as the mean rotational time between two random track locations (*i.e.*, the time it takes the disk to spin half a revolution).

Earlier studies [25, 28] have discovered that the seek time depends on the seek distance D_{seek} (distance to be traveled by the disk head) in the following way:

$$T_{seek} = \begin{cases} 0, & \text{if } D_{seek} = 0; \\ a + b\sqrt{\frac{D_{seek}}{D_{disk}}}, & \text{if } 0 < \frac{D_{seek}}{D_{disk}} \leq e; \\ c + d \cdot \frac{D_{seek}}{D_{disk}}, & \text{if } e < \frac{D_{seek}}{D_{disk}} \leq 1. \end{cases} \quad (14)$$

where D_{disk} is the total disk size. a, b, c, d, e are disk-specific parameters and $a + b\sqrt{e} \approx c + d \cdot e$.

Combining the seek distance distribution in Equation (7) and the above Equation (14), we have the following cumulative probability distribution for the seek time:

$$Pr[T_{seek} \geq x] = \begin{cases} 1, & \text{if } x \leq a; \\ \left(1 - \frac{(\frac{x-a}{b})^2}{e}\right)^\gamma, & \text{if } a < x \leq a + b\sqrt{e}; \\ \left(1 - \frac{(\frac{x-c}{d})}{e}\right)^\gamma, & \text{if } c + d \cdot e < x \leq c + d \cdot \delta; \\ 0, & \text{if } c + d \cdot \delta < x. \end{cases} \quad (15)$$

Therefore, the expected average seek time is:

$$\begin{aligned} E(T_{seek}) &= \int_0^a Pr[T_{seek} \geq x] dx + \int_a^{a+b\sqrt{e}} Pr[T_{seek} \geq x] dx \\ &\quad + \int_{c+d \cdot e}^{c+d \cdot \delta} Pr[T_{seek} \geq x] dx \\ &= a + \int_a^{a+b\sqrt{e}} \left(1 - \frac{(\frac{x-a}{b})^2}{e}\right)^\gamma dx + \int_{c+d \cdot e}^{c+d \cdot \delta} \left(1 - \frac{(\frac{x-c}{d})}{e}\right)^\gamma dx \\ &= a + b \cdot \int_0^{\sqrt{e}} \left(1 - \frac{x^2}{e}\right)^\gamma dx + d \cdot \int_e^\delta \left(1 - \frac{x}{e}\right)^\gamma dx \\ &= a + b\sqrt{e} \cdot \left(\sum_{i=0}^{\gamma} \frac{\binom{\gamma}{i}}{2i+1} \cdot \left(-\frac{e}{\delta}\right)^i\right) + \frac{d \cdot \delta}{\gamma+1} \cdot \left(1 - \frac{e}{\delta}\right)^{\gamma+1} \end{aligned}$$

where $\binom{\gamma}{i}$ indicates the binomial coefficient.

(16)

Disk drives are usually equipped with limited amount of cache. Due to its small size, its main usage is disk track prefetching while its caching effects are negligible for data-intensive applications with large working-set sizes. We do not consider such caching effects in our model.

3.4 Symbol Definitions

For clarity, we list the definitions for all symbols used in the previous subsections (Table 1).

3.5 Model Interfaces

We summarize the interfaces to our performance model, which include the workload characteristics, operating system configuration, and storage device properties.

- Table 2 lists the attributes of workload characteristics passed into our model. The table also lists the OS component in our performance model that is most concerned with each workload attribute.
- The OS configuration consists of the I/O prefetching depth and whether to employ the anticipatory I/O scheduler or the classic elevator scheduler.
- The storage device properties include the disk size, rotational speed, seek time model parameters of Equation (14), and the histogram of data transfer rate at each disk location.

<i>Symbol</i>	<i>Definition</i>
S_{stream}, S'_{stream}	original and transformed sequential access stream lengths
S_{run}	the sequential access run length
$S_{prefetch}$	I/O prefetching depth
$N_{prefetch}$	the number of I/O prefetching requests for accessing a stream
$S_{request}$	the I/O request size
γ	the number of concurrent request executions in the server
D_{seek}, D_{disk}	the seek distance and the total disk size
δ	the proportion of the dataset span to the total disk size
$T_{seek}, T_{rotation}, T_{idle}, T_{disk}$	the disk seek, rotation, idle, and total usage time
R_{tr}	the disk data transfer rate
a, b, c, d, e	disk-specific parameters concerning the disk seek time

Table 1: Definition of symbols used in Section 3.

<i>Workload attribute</i>	<i>Unit</i>	<i>Concerned OS component</i>
server concurrency	a number	I/O scheduling (Section 3.2)
data span on storage medium	ratio to the disk size	I/O scheduling (Section 3.2)
lengths of sequential access streams	a histogram	I/O prefetching (Section 3.1)
whether each stream access whole file	true or false	I/O prefetching (Section 3.1)
average sequential access run length	unit of data size	anticipatory I/O scheduling (Section 3.2.2)
average application thinktime	unit of time	anticipatory I/O scheduling (Section 3.2.2)

Table 2: Attributes of workload characteristics. We also list the OS component in our performance model that is most concerned with each workload attribute.

4 Model-driven Performance Debugging

Based on the whole-system performance model for I/O-intensive online servers, this section describes our approach to acquire a representative set of anomalous workload and configuration settings. We also present techniques to cluster anomalous settings into groups likely attributed to individual bugs. We then characterize each of them with correlated system component and workload conditions. Although certain low-level techniques in our approach are specifically designed for our target system and workloads, we believe the general framework of our approach can also be used for performance debugging of other large software systems.

4.1 Anomaly Sampling

Performance anomalies (manifested by deviations of measurement results from the model-predicted performance) occur for several reasons. In addition to performance bugs in the implementation, measurement errors and model inaccuracies can also cause performance anomalies. Aside from significant modeling errors, anomalies caused by these other factors are usually small in magnitude. We screen out these factors by only counting the relatively large performance anomalies. Although this screening may also overlook some performance bugs, those that cause significant performance degradations would not be affected.

Performance anomalies may occur at many different workload conditions and system configurations. We con-

sider each occurrence under one setting as a single point in the multi-dimensional space where each workload condition and system configuration parameter is represented by a dimension. For the rest of this paper, we call this multi-dimensional space simply as the *parameter space*. Our anomaly sampling proceeds in the following two steps. First, we choose a number of (n) sample settings from the parameter space in a uniformly random fashion. We then compare measured system performance with model prediction under these settings. Anomalous settings are those at which measured performance trails model prediction by at least a certain threshold.

We define the *infliction zone* of each performance bug as the union of settings in the parameter space at which the bug would inflict significant performance losses. By examining a uniformly random set of sample settings, our anomaly sampling approach can achieve the following property associated with false negatives (missing some bugs). For a bug whose infliction zone is p proportion ($0 < p \leq 1$) of the total parameter space, the probability for at least one of our n random samples falls into the bug's infliction zone is $1 - (1 - p)^n$. With a reasonably large n , it is unlikely for our anomaly sampling to miss a performance bug that takes effects under a non-trivial set of workload conditions and system configurations.

We now describe the parameter space for our target workload and system. We first explore the dimensions representing workload properties and we will examine the system configuration dimensions next.

Workload properties The inclusion of each workload property in the parameter space allows the characterization of its relationship with performance bugs in subsequent analysis. However, considering too many workload properties may render the subsequent analysis intractable. According to our performance model in Section 3, we select workload properties from those that have large effects on system performance. For each workload property, we determine several representative parameter settings for possible sampling.

- *Server concurrency*: 1, 2, 4, 8, 16, 32, 64, 128, 256.
- *Average length of sequential access streams*: 64 KB, 128 KB, 256 KB, 512 KB, 1 MB, 2 MB, 4 MB.
- *Whether each stream access whole file*: true or false.
- *Average length of sequential access runs*: 16 KB, 32 KB, 64 KB, \dots , up to the average length of sequential access streams.
- *Average application thinktime per megabyte of data access*: 1 ms, 2 ms, 4 ms, 8 ms.

For the purpose of real system measurement, we design an adjustable micro-benchmark that can exhibit any combination of workload parameter settings.

System configurations The inclusion of system configurations in the parameter space allows the characterization of their relationships with performance bugs in subsequent analysis. In particular, the strong correlation between a performance bug and the activation of a system component indicates the likelihood that the bug is within the implementation of the said component. As indicated in our performance model, the system performance is mainly affected by three I/O system components: prefetching, the elevator I/O scheduler and the anticipatory I/O scheduler.

For each system component that is considered for debugging, we must include system configurations where the component is not activated. The two I/O schedulers are natural alternatives to each other. We augment the operating system to add an option to bypass the prefetching code. We do so by ignoring the readahead heuristics and issuing I/O requests only when data is synchronously demanded by the application. Since our performance model does not consider OS caching, we also add additional code in the operating system to disable the caching. We do so by simply overlooking the cached pages during I/O. Our changes are only a few hundred lines in the Linux 2.6.10 kernel.

Below are the specific dimensions in our parameter space that represent system configurations:

- *Prefetching*: enabled or disabled.
- *I/O scheduling*: elevator or anticipatory.

Our performance model in Section 3 can predict system performance at different prefetching sizes. However, varying the prefetching size is not useful for our purpose of performance debugging. We use the default maximum prefetching size (128 KB for Linux 2.6.10) in our study.

4.2 Anomaly Clustering and Characterization

Given a set of anomalous workload condition and system configuration settings, it is still hard to derive useful debugging information without a succinct characterization on the anomalous settings. Further, the system may contain multiple independent performance bugs and the aggregate characteristics of several bugs may be too confusing to be useful. This section presents an algorithm to cluster anomalous settings into groups likely attributed to individual bugs and characterize each cluster to guide performance debugging. At a high level, the anomaly sampling described in Section 4.1 precedes the clustering and characterization, which are then followed by the final human debugging. Ideally, each such action sequence can discover one performance bug and multiple bugs can be identified by iterating this action sequence multiple times.

It is quite common for infliction zones of multiple bugs to cross-intersect with each other. In other words, several bugs might inflict performance losses simultaneously at a single workload condition and system configuration. Classical clustering algorithms such as Expectation Maximization (EM) [10] and K-means [19] typically assume disjoint (or slightly overlapped) clusters and spherical Gaussian distribution for points in each cluster. Therefore they cannot be directly used to solve our problem.

To make our clustering problem more tractable, we assume that the infliction zone of each performance bug takes a hyper-rectangle-like shape in the parameter space. This means that if parameter settings (a_1, a_2, \dots, a_k) and (b_1, b_2, \dots, b_k) in the k -dimensional parameter space are inflicted by a bug, then any parameter setting (x_1, x_2, \dots, x_k) with

$$\begin{cases} a_1 \leq x_1 \leq b_1 \\ a_2 \leq x_2 \leq b_2 \\ \dots\dots\dots \\ a_k \leq x_k \leq b_k \end{cases} \quad (17)$$

also likely falls into the bug's infliction zone. For each dimension i that has no ordering among its value settings (e.g., a boolean or categorical parameter), the corresponding element in Condition (17) should be instead " $x_i = a_i$ or $x_i = b_i$ ".

A bug's infliction zone takes a hyper-rectangle-like shape if it has a range of triggering settings on each parameter (workload property or system configuration)

and the bug's performance effect is strongly correlated with the condition that all parameters fall into respective triggering ranges. When this assumption does not hold for a bug (*i.e.*, its infliction zone does not follow a hyper-rectangle-like shape), our algorithm described below would identify a maximum hyper-rectangle encapsulated within the bug's infliction zone. This might still provide some useful bug characterization for subsequent human debugging.

To the best of our knowledge, the only known clustering algorithm that handles intersected hyper-rectangles is due to Pelleg and Moore [21]. However, their algorithm requires hyper-rectangles to have *soft* boundaries with Gaussian distributions and hence is not directly applicable to our case, where hyper-rectangles could have infinitely steeply diminishing borders.

We describe our algorithm that identifies and characterizes one dominant cluster from a set of anomalous settings. More specifically, our algorithm attempts to identify a hyper-rectangle in the parameter space that explores trade-off between two properties: 1) Most of the sample settings within the hyper-rectangle are anomalous settings; 2) The hyper-rectangle contains as many anomalous settings as possible. In our algorithm, property 1 is ensured by keeping the ratio of $\frac{\# \text{ of anomalies}}{\# \text{ of samples}}$ in the hyper-rectangle above a certain pre-defined threshold. Property 2 is addressed by greedily expanding the current hyper-rectangle in a way to maximize the number of anomalous settings contained in the expanded new hyper-rectangle. Algorithm 4.1 illustrates our method to discover a hyper-rectangle that tightly bounds the cluster of anomalous settings related to a dominant bug.

After the hyper-rectangle clustering, we characterize each cluster by simply projecting the hyper-rectangle onto each dimension of the parameter space. For each dimension (a workload property or a system configuration), we include the projected parameter value range into the characterization. For those dimensions at which the projections cover all possible parameter values, we consider them uncorrelated to the cluster and we do not include them in the cluster characterization.

The computation complexity of our algorithm is $O(m^3n)$ since the algorithm has three nested loops with at most m iterations for each. In the innermost loop, the numbers of samples and anomalies within a hyper-rectangle are computed by brute-force checking of all n sample settings (an $O(n)$ complexity). Using pre-constructed orthogonal range trees [18], the complexity of the innermost loop can be improved to $O((\log n)^d + A)$, where d is the dimensionality of the parameter space and A is the answer size. We use brute-force counting in our current implementation due to its simplicity and satisfactory performance on our dataset (no more than 1000 sample settings and less than 200 anomalies).

Algorithm 4.1: CLUSTER(n samples, m anomalies, ϵ)

Input: n sample settings.

Input: m anomalous settings among the samples.

Input: $0 < \epsilon \leq 1$, the threshold for $r(H)$.

Returns: H_{max} , a hyper-rectangle in the parameter space.

$H_{max} \leftarrow nil$

for each x out of m anomalous settings

$H \leftarrow$ the min-bounding hyper-rectangle for x

while H was just expanded

$y_{tmp} \leftarrow nil$
 $c_{tmp} \leftarrow 0$

for each anomalous setting y outside H

do $\left\{ \begin{array}{l} \text{if } [r(M(H, y)) \geq \epsilon \\ \text{and } c(M(H, y)) > c_{tmp}] \\ \text{then } \left\{ \begin{array}{l} y_{tmp} \leftarrow y \\ c_{tmp} \leftarrow c(M(H, y)) \end{array} \right. \end{array} \right.$

if $[y_{tmp} \neq nil]$
 then $H \leftarrow M(H, y_{tmp})$

if $[c(H) > c(H_{max})]$
 then $H_{max} \leftarrow H$

return (H_{max})

/ $r(H)$ denotes the ratio of $\frac{\# \text{ of anomalies}}{\# \text{ of samples}}$ in the hyper-rectangle H .*

$c(H)$ denotes the number of anomalies in H .

$M(H, y)$ denotes the minimum-bounding hyper-rectangle that contains the hyper-rectangle H and the point y . */

5 Debugging Results

We describe our performance debugging of the Linux 2.6.10 kernel (released in December 2004) when supporting I/O-intensive online servers. We repeatedly perform anomaly sampling, clustering, characterization, and human debugging. After each round, we acquire an anomaly cluster characterization that corresponds to one likely bug. The characterization typically contains correlated system component and workload conditions, which hints at where and how to look for the bug. The human debugger has knowledge on the general structure of the OS source code and is familiar with a kernel tracing tool (LTT [37]). After each bug fix, we use the corrected kernel for the next round of anomaly sampling, clustering, characterization, and human debugging.

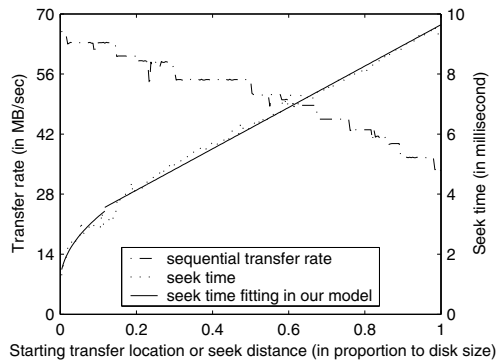


Figure 4: Data transfer rate and seek time curve for the disk drive. We also show the seek time fitting used in Equation (14) of our performance model.

Our measurement uses a server equipped with dual 2.0GHz Xeon processors, 2GB memory, and an IBM 10KRPM SCSI drive (model "DTN036C1UCDY10"). We measure the disk drive properties as input to our performance model (shown in Figure 4). The Equation (14) parameters for this disk is $a=1.0546$ ms, $b=6.9555$ ms, $c=2.7539$ ms, $d=6.8867$ ms, and $e=0.1171$. We choose 400 random workload and system configuration settings in the anomaly sampling. The anomaly threshold is set at 10% (i.e., those settings at which measured performance trails model prediction by at least 10% are considered as anomalous settings). The clustering threshold (ϵ) in Algorithm 4.1 is set at 90%.

We describe our results below and we also report the debugging time at the end of this section. The first anomaly cluster characterization is:

Workload property	
<i>Concurrency:</i>	128 and above
<i>Stream length:</i>	256KB and above
System configuration	
<i>Prefetching:</i>	enabled

This characterization shows that the corresponding bug concerns the prefetching implementation and it inflicts performance losses for high concurrency workloads with moderately long sequential access streams. Based on this information, our subsequent tracing and analysis discover the following performance bug. The kernel checks for disk congestion when each I/O prefetching is initiated. If the number of pending requests in the disk driver queue exceeds a certain threshold (slightly below 128 in Linux 2.6.10), the prefetching is canceled. The intuition for this treatment is that asynchronous read-ahead should be disabled when the I/O system is busy. However, the prefetching operations may include some data that is synchronously demanded by the application. By canceling these operations, it causes confusion at upper-level I/O code and results in inefficient single-page makeup I/Os

for the needed data. In order to fix this problem, the corrected kernel only cancels prefetching requests that do not contain any synchronously demanded data when disk congestion occurs. We call this *bug fix #1*.

The second anomaly cluster characterization is:

Workload property	
<i>Concurrency:</i>	8 and above
<i>Stream length:</i>	256KB and above
<i>Run length:</i>	256KB and above
System configuration	
<i>I/O scheduling:</i>	anticipatory

This characterization concerns the anticipatory I/O scheduler. It involves workloads at moderately high concurrency with stream and run lengths larger than the maximum prefetching size (128 KB). Our subsequent investigation discovers the following performance bug. The current implementation of the anticipatory scheduler stops an ongoing anticipation if there exists a pending I/O request with shorter seek distance (compared with the average seek distance of the anticipating process). Due to a significant seek initiation cost on modern disks (as shown in Figure 4), the seek distance is not an accurate indication of the seek time cost. For example, the average cost of a 0-distance seek and a $2x$ -distance seek is much less than an x -distance seek. As the result, the current implementation tends to stop the anticipation when the benefit of continued anticipation actually exceeds that of breaking it. We solve this problem by using estimated seek time (instead of the seek distance) in the anticipation cost/benefit analysis. We call this *bug fix #2*.

The third anomaly cluster characterization is:

Workload property	
<i>Concurrency:</i>	2
System configuration	
<i>I/O scheduling:</i>	elevator

This characterization concerns the elevator scheduler (also called the deadline scheduler in Linux 2.6.10) and the corresponding bug inflicts performance losses at the concurrency of 2. Our tracing and analysis show that a *reset* function is called frequently at very low concurrency. Possibly due to an overly-simplified implementation, the kernel always searches from block address 0 for the next scheduled request after the reset. We fix it by searching from the last I/O location according to the elevator scheduling algorithm. We call this *bug fix #3*.

The fourth anomaly cluster characterization is:

Workload property	
<i>Concurrency:</i>	2 and above
<i>Stream length:</i>	256KB and above
<i>Run length:</i>	256KB
System configuration	
<i>I/O scheduling:</i>	anticipatory

This characterization concerns the anticipatory I/O scheduler for non-serial concurrent workloads. Our subsequent investigation uncovers the following problem. Large I/O requests (including maximum-sized prefetching requests) from the file system are often split into smaller pieces before being forwarded to the disk drive. The completion of each one of these pieces will trigger an I/O interrupt. The original anticipatory scheduler would start the anticipation timer right after the first such interrupt, which often causes premature timeout. We correct the problem by starting the anticipation timer only after all pieces of a file system I/O request have completed. We call this *bug fix #4*.

We show results on the effects of our bug fixes. Figure 5 shows the top 10% model/measurement errors of our anomaly sampling for the original Linux 2.6.10 kernel and after the accumulative bug fixes. The error is defined as $1 - \frac{\text{measured throughput}}{\text{model-predicted throughput}}$. Results show that performance anomalies steadily decrease after each bug fix and no anomaly with 14% or larger error exists after all four bugs are fixed. Figure 6 illustrates all-sample comparison between model prediction and measured performance. Figure 6(A) shows results for the original Linux 2.6.10 where the system performs significantly worse than model prediction at many parameter settings. Figure 6(B) shows the results when all four bugs are fixed where the system performs close to model prediction at all parameter settings.

Debugging time We provide statistics on the debugging time. For each bug fix, time is spent on anomaly sampling, clustering and characterization, and final human debugging.

- The primary time cost for anomaly sampling is on the system measurement for all sample workload condition and system configuration settings. The measurement of each sample setting took around 6 minutes and the total 400 sample measurements took around two days using one test server. More test servers would speed up this process proportionally.
- Due to the relative small sample size, our clustering and characterization algorithm took less than a minute to complete.
- The final human debugging took about one or two days for each bug fix.

6 Evaluation with Real Workloads

We experiment with real server workloads to demonstrate the performance benefits of our bug fixes. All measurements are conducted on servers each equipped with

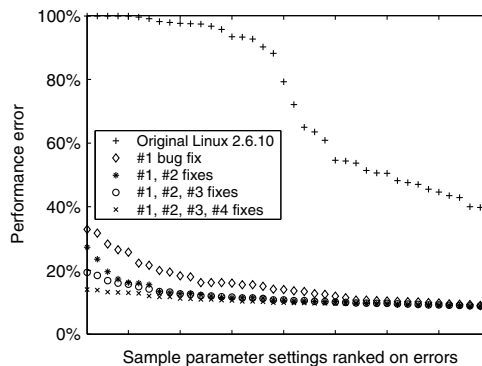


Figure 5: Top 10% model/measurement errors. Each unit on the X-axis represents a sampled parameter setting in our anomaly sampling.

dual 2.0GHz Xeon processors, 2 GB memory, and an IBM 10 KRPM SCSI drive (as characterized in Figure 4). Each experiment involves a server and a load generation client. The client can adjust the number of simultaneous requests to control the server concurrency level.

6.1 Workload Descriptions

We evaluate four server workloads in our study:

- *SPECweb99*: We include the SPECweb99 benchmark [30] running on the Apache 2.0.44 Web server. This workload contains 4 classes of files with sizes at 1 KB, 10 KB, 100 KB, and 1,000 KB respectively. During each run, the four classes of files are accessed according to a distribution that favors small files. Within each class, a Zipf distribution with parameter $\alpha = 1.0$ is used to access individual files.
- *Media clips*: Web workloads such as SPECweb99 contain mostly small file accesses. In order to examine the effects of relatively large sequential access streams, we use a Web workload containing a set of media clips, following the file size and access distribution of the video/audio clips portion of the 1998 World Cup workload [3]. About 67% (in total size) of files in the workload are large video clips, while the rest are small audio clips. The file sizes of both small and large clips follow Lognormal distributions, with average sizes of 20 KB and 1,464 KB respectively. During the tests, individual media files are chosen as client requests in a uniformly random fashion.
- *Index searching*: We acquired a prototype of the index searching server and a dataset from the Web search engine Ask Jeeves [4]. The dataset contains the search index for 12.6 million Web pages. It includes a mapping file that maps MD5-encoded keywords to proper locations in the search index. For

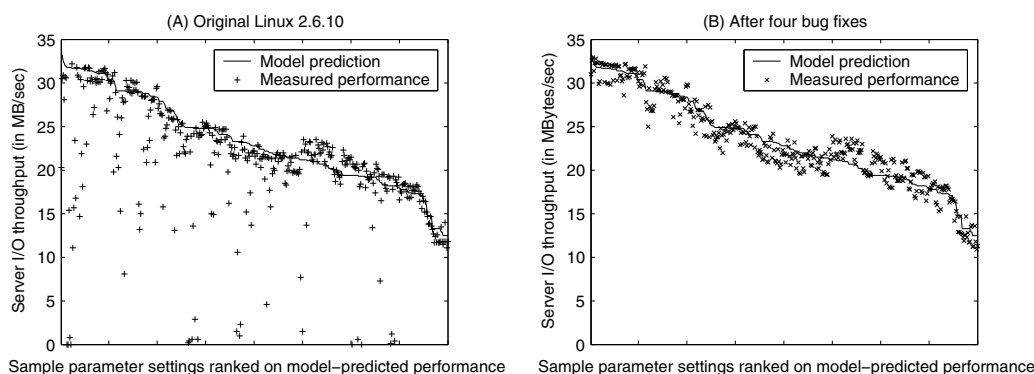


Figure 6: All-sample comparison between model prediction and measured performance. Each unit on the X-axis represents a sampled parameter setting in our anomaly sampling.

Workload	Data size	Data popularity	Whole file access	Mean stream len.	Runs/stream	Thinktime/MB
SPECweb99	22.4 GB	Zipf	yes	67.2 KB	1.00	1.11 ms
Media clips	27.2 GB	Uniformly random	yes	1213.3 KB	1.01	1.78 ms
Index searching	18.5 GB	Trace-driven	no	267.2 KB	1.75	0.22 ms
TPC-C	8.8 GB	Uniformly random	no	43.8 KB	1.00	11.69 ms

Table 3: Characteristics of four server workloads used in our evaluation.

each keyword in an input query, a binary search is first performed on the mapping file and then the search index is accessed following a sequential access pattern. Multiple prefetching streams on the search index are accessed for each multi-keyword query. The search query words in our test workload are based on a one-week trace recorded at the Ask Jeeves site in early 2002.

- *TPC-C database benchmark:* We include a local implementation of the TPC-C online transaction processing benchmark [32] in our evaluation. TPC-C simulates a population of terminal operators executing Order-Entry transactions against a database. Some of the TPC-C transactions do not consume much I/O resource. We use a workload that contains only the “new order” transactions, which are the most I/O-intensive among five types of TPC-C transactions. In our experiments, the TPC-C benchmark runs on the MySQL 5.0.2-alpha database with a dataset of 8.8 GB.

To better understand these workloads, we extract their characteristics through profiling. During profiling runs, we intercept relevant I/O system calls in the OS kernel, including open, close, read, write, and seek. We extract desired application characteristics after analyzing the system call traces collected during profiling runs. However, system call interception does not work well for memory mapped I/O used by the TPC-C database. In this case, we intercept device driver-level I/O traces and use them to infer the data access pattern of the workload. Table 3 lists some characteristics of the four server

workloads. The stream statistics for TPC-C are for read streams only. Among the four workloads, we observe that media clips has long sequential access streams while SPECweb99 and TPC-C have relatively short streams. We also observe that the three workloads except the index searching have about one run per stream, which indicates that each request handler does not perform interleaving I/O when accessing a sequential stream.

6.2 Performance Results

Figure 7 illustrates the throughput of the four server workloads. For each workload, we show measured performance at different concurrency levels under the original Linux kernel and after various performance bug fixes. The elevator I/O scheduler is employed for SPECweb99 and media clips while the anticipatory I/O scheduler is used for index searching and TPC-C. Therefore bug fix #3 is only meaningful for SPECweb99 and media clips while fixes #2 and #4 are only useful for index searching and TPC-C. The I/O throughput results are those observed at the application level. They are acquired by instrumenting the server applications with statistics-collection code. We were not able to make such instrumentation for the MySQL database used by TPC-C so we only show the request throughput for this workload.

Suggested by the characterization of bug #1, Figure 7(B) and (C) confirm substantial performance improvement (around five-fold) of the bug fix at high execution concurrencies. We notice that its effect is not as obvious for SPECweb99 and TPC-C. This can also be explained by our characterization of bug #1 since these

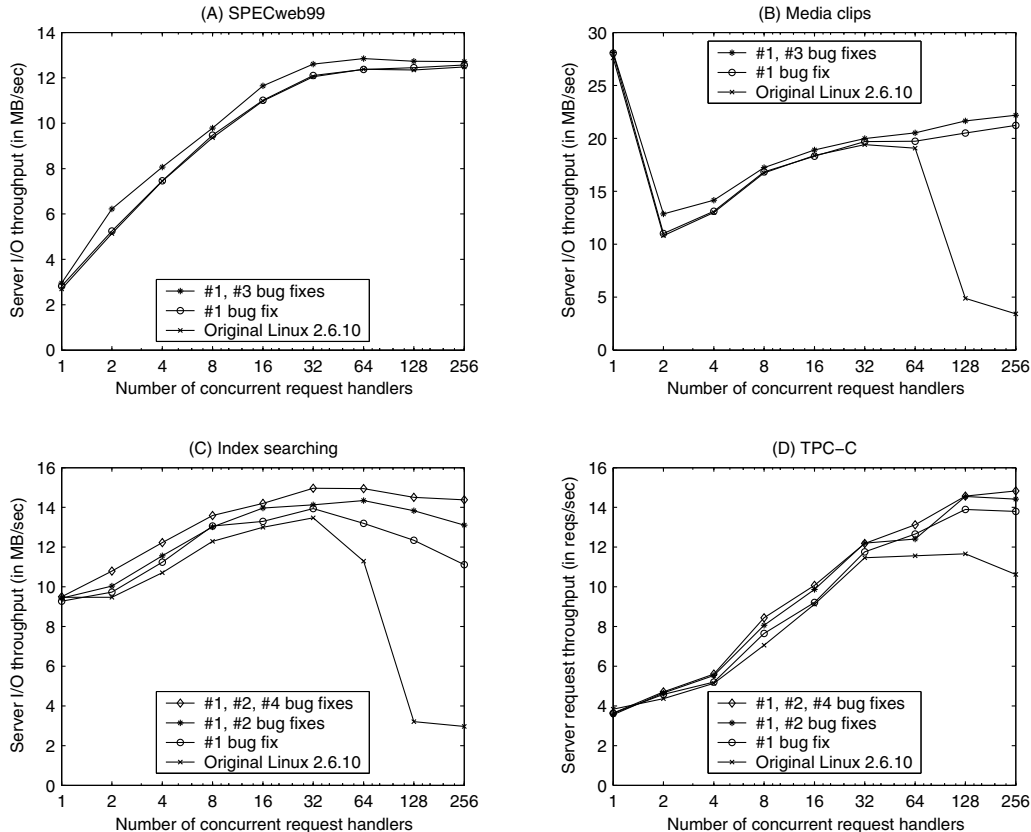


Figure 7: Throughput of four server workloads under various kernels. The elevator I/O scheduler is employed for SPECweb99 and media clips while the anticipatory I/O scheduler is used for index searching and TPC-C.

workloads do not have long enough sequential access streams. The other bug fixes provide moderate performance enhancement for workloads that they affect. The average improvement (over all affected workload conditions) is 6%, 13%, and 4% for bug fix #2, #3, and #4 respectively.

Aggregating the effects of all bug fixes, the average improvement (over all tested concurrencies) of the corrected kernel over the original kernel is 6%, 32%, 39%, and 16% for the four server workloads respectively.

7 Related Work

Performance debugging. Earlier studies have proposed techniques such as program instrumentation (*e.g.*, MemSpy [20] and Mtool [13]), complete system simulation (*e.g.*, SimOS [24]), performance assertion checking [22], and detailed overhead categorization [9] to understand performance problems in computer systems and applications. These techniques focus on offering fine-grained examination of the target system/application in specific workload settings. Many of them are too expensive to be used for exploring wide ranges of workload conditions and system configurations. In compari-

son, our approach trades off detailed execution statistics at specific settings for comprehensive characterization of performance anomalies over wide ranges of workloads.

Recent performance debugging work employs statistical analysis of online system traces [1, 7] to identify faulty components in complex systems. Such techniques are limited to reacting to anomalies under past and present operational environments and they cannot be used to debug a system before such operational conditions are known. Further, our approach can provide the additional information of correlated workload conditions with each potential performance bug, which is helpful to the debugging process.

Identifying non-performance bugs in complex systems. Several recent works investigated techniques to discover non-performance bugs in large software systems. Engler *et al.* detect potential bugs by identifying anomalous code that deviates from the common pattern [11]. Wang *et al.* discover erroneous system configuration settings by matching with a set of known correct configurations [34]. Li *et al.* employ data mining techniques to identify copy-paste and related bugs in operating system code [17]. However, performance-oriented debugging can be more challenging because many performance bugs are strongly connected with the code se-

mantics and they often do not follow certain patterns. Further, performance bugs may not cause obvious misbehaviors such as incorrect states or system crashes. Without an understanding on the expected performance (e.g., through the performance model that we built), it may not even be easy to tell the existence of performance anomalies in complex systems.

I/O system performance modeling. Our performance debugging approach requires the construction of a whole-system performance model for targeted I/O-intensive server workloads. A large body of previous studies have constructed various analytical and simulation models to examine the performance of storage and I/O systems, including those for disk drives [5, 16, 25, 28, 36], disk arrays [2, 8, 33], I/O scheduling algorithms [23, 26, 35], and I/O prefetching [6, 29, 31]. However, performance models for individual system components do not capture the interplay between different components. This paper presents a whole-system throughput model that considers the combined impact of the application characteristics and several relevant operating system components on the overall server performance.

Using system-level models to predict the performance of I/O-intensive workloads is not new. Ganger and Patt argued that the I/O subsystem model must consider the *criticality* of I/O requests, which is determined by application and OS behaviors [12]. Shriver *et al.* studied I/O system performance using a combined disk and OS prefetching model [29]. However, these models do not consider recently proposed I/O system features. In particular, we are not aware of any prior I/O system modeling work that considers the anticipatory I/O scheduling, which can significantly affect the performance of our targeted workloads.

8 Conclusion

This paper presents a new performance debugging approach for complex software systems using model-driven anomaly characterization. In our approach, we first construct a whole-system performance model according to the design protocol/algorithms of the target system. We then acquire a representative set of anomalous workload settings by comparing measured system performance with model prediction under a number of sample settings. We statistically cluster the anomalous settings into groups likely attributed to individual bugs and characterize them with specific system components and workload conditions. Compared with previous performance debugging techniques, the key advantage of our approach is that we can comprehensively characterize performance anomalies of a complex system under wide ranges of workload conditions and system configurations.

We employ our approach to quickly identify four performance bugs in the I/O system of the recent Linux 2.6.10 kernel. Our anomaly characterization provides hints on the likely system component each performance bug may be located at and workload conditions for the bug to inflict significant performance losses. Experimental results demonstrate substantial performance benefits of our bug fixes on four real server workloads.

Acknowledgments We benefited greatly from Athanasios Papathanasiou's expertise in Linux kernel development and particularly his help in identifying the cause for the first bug described in Section 5. We would also like to thank Christopher Stewart, Yuan Sun, and the anonymous referees for helpful discussions and valuable comments during the course of this work.

References

- [1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proc. of the 19th ACM SOSP*, pages 74–89, Bolton Landing, NY, October 2003.
- [2] E. Anderson, R. Swaminathan, A. Veitch, G. A. Alvarez, and J. Wilkes. Selecting RAID Levels for Disk Arrays. In *Proc. of the 1st USENIX Conf. on File and Storage Technologies*, pages 189–201, Monterey, CA, January 2002.
- [3] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35, HP Laboratories Palo Alto, 1999.
- [4] Ask Jeeves Search. <http://www.ask.com>.
- [5] R. Barve, E. Shriver, P. B. Gibbons, B. K. Hillyer, Y. Matias, and J. S. Vitter. Modeling and Optimizing I/O Throughput of Multiple Disks on A Bus. In *Proc. of the ACM SIGMETRICS*, pages 83–92, Atlanta, GA, June 1999.
- [6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proc. of the ACM SIGMETRICS*, pages 188–197, Ottawa, Canada, June 1995.
- [7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. In *Proc. of Int'l Conf. on Dependable Systems and Networks*, pages 595–604, Washington, DC, June 2002.
- [8] P. M. Chen, G. A. Gibson, R. H. Katz, and D. A. Patterson. An Evaluation of Redundant Arrays of Disks using an Amdahl 5890. In *Proc. of the ACM SIGMETRICS*, pages 74–85, Boulder, CO, May 1990.
- [9] M. E. Crovella and T. J. LeBlanc. Parallel Performance Prediction Using Lost Cycles Analysis. In *Proc. of Super-Computing*, pages 600–610, Washington, DC, November 1994.

- [10] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data Via the EM Algorithm. *Journal of the Royal Statistical Society, Series B*, (1):1–38, 1977.
- [11] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proc. of the 18th ACM SOSOP*, pages 57–72, Banff, Canada, October 2001.
- [12] G. R. Ganger and Y. N. Patt. Using System-Level Models to Evaluate I/O Subsystem Designs. *IEEE Trans. on Computers*, 47(6):667–678, June 1998.
- [13] A. J. Goldberg and J. L. Hennessy. Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Trans. on Parallel and Distributed Systems*, 4(1):28–40, January 1993.
- [14] S. Iyer and P. Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of the 18th ACM SOSOP*, pages 117–130, Banff, Canada, October 2001.
- [15] P. Jelenkovic and A. Radovanovic. The Persistent-Access-Caching Algorithm. Technical Report EE-2004-03-05, Dept. of Electrical Engineering, Columbia University, 2004.
- [16] D. Kotz, S. B. Toh, and S. Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical Report PCS-TR94-220, Dept. of Computer Science, Dartmouth College, July 1994.
- [17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *Proc. of the 6th USENIX OSDI*, pages 289–302, San Francisco, CA, December 2004.
- [18] G. S. Lueker. A Data Structure for Orthogonal Range Queries. In *Proc. of the 19th IEEE Symp. on Foundations of Computer Science*, pages 28–34, 1978.
- [19] J. B. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proc. of the 5th Berkeley Symp. on Mathematical Statistics and Probability*, pages 281–297, 1967.
- [20] M. Martonosi, A. Gupta, and T. Anderson. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *Proc. of the ACM SIGMETRICS*, pages 1–12, Newport, RI, June 1992.
- [21] D. Pelleg and A. Moore. Mixtures of Rectangles: Interpretable Soft Clustering. In *Proc. of the 18th Int’l Conf. on Machine Learning*, pages 401–408, Berkshires, MA, June 2001.
- [22] S. E. Perl and W. E. Weihl. Performance Assertion Checking. In *Proc. of the 14th ACM SOSOP*, pages 134–145, Asheville, NC, December 1993.
- [23] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proc. of the USENIX Annual Technical Conf.*, pages 297–310, San Antonio, TX, June 2003.
- [24] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Trans. on Modeling and Computer Simulation*, 7(1):78–103, January 1997.
- [25] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.
- [26] P. J. Shenoy and H. M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proc. of the ACM SIGMETRICS*, pages 44–55, Madison, WI, June 1998.
- [27] E. Shriver. *Performance Modeling for Realistic Storage Devices*. PhD thesis, Dept of Computer Science, New York University, 1997.
- [28] E. Shriver, A. Merchant, and J. Wilkes. An Analytical Behavior Model for Disk Drives with Readahead Caches and Request Reordering. In *Proc. of the ACM SIGMETRICS*, pages 182–192, Madison, WI, June 1998.
- [29] E. Shriver, C. Small, and K. A. Smith. Why Does File System Prefetching Work? In *Proc. of the USENIX Annual Technical Conf.*, pages 71–84, Monterey, CA, June 1999.
- [30] SPECweb99 Benchmark. <http://www.specbench.org/osg/web99>.
- [31] A. Tomkins, R. H. Patterson, and G. A. Gibson. Informed Multi-Process Prefetching and Caching. In *Proc. of the ACM SIGMETRICS*, pages 100–114, Seattle, WA, June 1997.
- [32] Transaction Processing Performance Council. TPC Benchmark C, Revision 5.4, April 2005. <http://www.tpc.org/tpcc/>.
- [33] M. Uysal, G. A. Alvarez, and A. Merchant. A Modular, Analytical Throughput Model for Modern Disk Arrays. In *Proc. of the 9th MASCOTS*, pages 183–192, Cincinnati, OH, August 2001.
- [34] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proc. of the 6th USENIX OSDI*, pages 245–258, San Francisco, CA, December 2004.
- [35] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proc. of the ACM SIGMETRICS*, pages 241–251, Santa Clara, CA, May 1994.
- [36] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proc. of the ACM SIGMETRICS*, pages 146–156, Ottawa, Canada, June 1995.
- [37] K. Yaghmour and M. R. Dagenais. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proc. of the USENIX Annual Technical Conf.*, San Diego, CA, June 2000.

TBBT: Scalable and Accurate Trace Replay for File Server Evaluation

Ningning Zhu, Jiawu Chen and Tzi-Cker Chiueh
Stony Brook University, Stony Brook, NY 11790
{nzhu, jiawu, chiueh}@cs.sunysb.edu

Abstract

This paper describes the design, implementation, and evaluation of TBBT, the first comprehensive NFS trace replay tool. Given an NFS trace, TBBT automatically detects and repairs missing operations in the trace, derives a file system image required to successfully replay the trace, ages the file system image appropriately, initializes the file server under test with that image, and finally drives the file server with a workload that is derived from replaying the trace according to user-specified parameters. TBBT can scale a trace temporally or spatially to meet the need of a simulation run without violating dependencies among file system operations in the trace.

1 Introduction

Modern file systems are typically optimized to take advantage of specific workloads. Therefore the performance of a file system must be evaluated with respect to its target workload. The ideal benchmarking workload should be representative of the way that actual applications will use the file system, effective in predicting the system's performance in the target environment, scalable so as to simulate the system under different loads, easy to generate, and reproducible.

At present, the most common workloads for file system evaluation are synthetic benchmarks. These benchmarks are designed to recreate the characteristics of particular environments. Although in recent years synthetic benchmarks have improved significantly in terms of realism and the degree with which they can be tailored to a specific application, it is not always possible for a synthetic benchmark to mimic file access traces collected from a real-world environment because there may be many time-varying and site-specific factors that are difficult, if not impossible, for a synthetic benchmark to capture. For example, recent file access trace analyses show that modern file servers are experiencing a variety of workloads with widely divergent characteristics [7, 18, 22]. Because the time required to develop a high-

quality benchmark is often on the order of months or years, benchmarks cannot keep up with the changes in the workload of specific target environments.

In contrast to synthetic benchmarks, traces taken from the system being evaluated are, by definition, representative of that system's workload. Trace replay can serve as a basis for file system workload generation and thereby performance evaluation. Although file system traces are often used as the basis for workload characterization and development of many file system design techniques, they are rarely used in the evaluation of file systems or servers. Given that disk, network, and web access traces have been used extensively to evaluate storage systems, network protocols, and web servers respectively, we see no reason why file access traces should not be a complementary method to evaluate file systems.

Replaying an NFS trace against a live file system/server is non-trivial for the following reasons. First, because a file system is stateful, a trace replay tool must ensure that the correct context for each request exists in the file system under test before it is replayed. For example, a file open request can be successfully replayed only if the associated file already exists. Second, the effect of aging on a file system may have a significant impact on its performance, and realistically and efficiently aging a file system is a difficult problem [20]. Finally, because a trace could be collected on a file system whose performance is very different from that of the target file system, it is essential that a trace replay tool be able to scale up or down the dispatch rate of trace requests to meet specific benchmarking requirements, without violating any inter-request dependencies.

In this paper we present the design, implementation, and evaluation of a novel NFS trace player named TBBT (Trace-Based file system Benchmarking Tool) and describe how it addresses each of these three issues. TBBT can infer from a trace the directory hierarchy of the file system underlying the trace, construct a file system image with the same hierarchy, replay the trace at a user-specified rate and gather performance measurements. Because traces do not carry physical layout in-

formation, it is impossible for TBBT to incorporate the actual aging effects in the construction of the initial file system image. However, TBBT does support an artificial aging method that allows users to incorporate a particular degree of file aging into the initial file system image used in their simulation. TBBT also allows its users to scale up the trace to simulate additional clients and/or higher-speed clients without violating the dependencies among file access requests in the trace. Finally, TBBT is robust in the face of tracing errors in that it can automatically detect and repair inconsistencies in incompletely collected traces.

TBBT is designed to overcome certain limitations of synthetic benchmarks, but it also has its own limitations, as described in Section 6. It is not meant to replace synthetic benchmarks but to complement them.

2 Related Work

Ousterhout's file system trace analysis [17] and the Sprite trace analysis [3] motivated many research efforts in log-structured file systems, journaling, and distributed file systems. More recent trace studies have demonstrated that file system workloads are diverse and vary widely depending on the applications they serve, and that workloads have changed over time, and raise new issues for researchers to address: Roselli *et al.* measured a range of workloads and showed that file sizes have become larger and large files are often accessed randomly, in contrast to findings from earlier studies [18]. Vogels showed that workloads on personal computers differ from most previously studied workloads [22]. More recently, Ellard and Mesnier *et al.* demonstrated that there is a strong relationship between the names and other create-time attributes and the lifespan, size and access patterns of files [7, 16].

Gibson *et al.* used a trace replay approach to evaluate two networked storage architectures: Networked SCSI disks and NASD [10]. Two traces were used: one is a week-long NFS trace from Berkeley [6] and the other is a month-long AFS trace from CMU. The traces were decomposed into *client-minutes*, each of which represents one minute of activity from a client. Specific *client-minutes* are selected, mixed and scaled to represent different workloads. Their paper did not mention how they perform file system initialization or handle dependency issues. Rather than implementing a full-fledged and accurate trace replay mechanism, their trace play tool is limited to the functionality required for their research.

There are two types of synthetic benchmarks. The first type generates a workload by using real applications. The examples include the Andrew Benchmark

[11], SSH-Build [19], and SDET [9]. The advantage of such benchmarks is that they capture the application dependencies between file system operations as well as application think-time. The disadvantage is that they are usually small and do not represent the workload of a large, general purpose networked file server.

The second type of synthetic benchmarks directly generate a workload through the system call interface or network file system protocol. Examples of such benchmarks include SPECsfs [21] and Postmark [12]. These benchmarks are easy to scale and fairly general-purpose, but it is difficult for such benchmarks to simulate a diverse and changing workload or the application-level operation dependencies and think-time. Recent research on file system benchmarking focuses on building flexible synthetic benchmarks to give user control over the workload patterns or building more complex models to emulate dependencies among file system operations (hBench [5], Fstress [1], FileBench [13]). In this paper, the main comparison of our work is with SPECsfs, a widely-used general-purpose benchmark for NFS servers [21]. Both SPECsfs and TBBT bypass the NFS client and access the server directly. SPECsfs attempts to recreate a typical workload based on characterization of real traces. Unfortunately, the result does not resemble any NFS workload we have observed. Furthermore, we question whether a typical workload actually exists – each NFS trace we have examined has unique characteristics.

Smith developed an artificial aging technique to create an aged file system image by running a workload designed to simulate the aging process [20]. This workload is created from file system snapshots and traces. After aging stage, there would be workload for benchmark run. The aging workload and the benchmark run workload have disjoint data sets. Yet since they share the same file system space, the free space fragmented by the aging workload would affect the benchmark run. This technique can be used for benchmarks that have a relatively small data set and do not have a dedicated initialization phase. Usually these benchmarks are micro-benchmarks or small macro-benchmarks such as SSH-Build. This technique is not applicable for benchmarks that take full control of a logical partition and has its own initialization procedure, such as SPECsfs. Smith's aging technique requires writing 80 GB of data (which requires several hours of run time) to age a 1 GB file system for the equivalent of seven months. This makes it impractical to use this method to age large file systems. TBBT ages the benchmark run load directly. TBBT's aging technique is less realistic, but runs two orders of magnitude more quickly.

Buttress is an disk I/O generation tool specifically designed to issue requests with accurate timing [2]. In

Field	Description
callTime	Timestamp of the request
replyTime	Timestamp of the reply
opType	NFS operation type
opParams	Request parameters (specific to the opType)
opReturn	Values returned by the operation

Table 1: Fields of a TBBT trace record.

benchmarking disk I/O systems, it is important to generate the I/O accesses that meet exactly the timing requirement. However, timing accuracy (issuing I/Os at the desired time) at high I/O rate is difficult to achieve on stock operating systems. TBBT suffers the same problem when the *timestamp-based* timing policy (as described in Section 3.4.1) is used to generate file system requests. Buttruss can generate I/O workloads with microsecond accuracy at the I/O throughput of high-end enterprise storage arrays. Buttruss’s timing control technique is flexible and portable, and provides a simple interface for load generation. TBBT could incorporate Buttruss’s technique to improve the timing accuracy of its request dispatching. There are other disk-level benchmarks such as IObench [23] and Imbench [15]. All disk-level benchmarks do not need to address the complicated issue of dependencies among file system operations.

3 Design Issues

3.1 Trace Transformation

TBBT uses a trace format that consists of a pair of request and reply - `<callTime, replyTime, opType, opParams, opReturn>`, as described in Table 1. The request and reply are paired up through their RPC message exchange ID. The `opType` is equivalent to the NFS procedure number in the original trace. The `opParams` and `opReturn` are similar to the corresponding NFS procedure parameters and return values. TBBT currently handles NFSv2 and NFSv3.

One important aspect of the TBBT trace format is creating the TBBT trace is more than a matter of simply reformatting the original trace. One example of this is the way that TBBT rewrites each NFS *filehandle*. In the NFS protocol, a *filehandle* is used to identify a specific file or directory. The problem is that it is possible for a single object to have more than one *filehandle* (because many implementations embed information such as the object version number and file system mount point inside the *filehandle*). To make matters worse, some NFS operations (such as `create`, `lookup`, and `remove`) use a name to identify a file instead of using a *filehandle*.

For example, in the case of `create` or `mkdir`, the *filehandle* is not known to the client because the file or directory does not yet exist. To avoid any potential for ambiguity, TBBT assigns TBBT-IDs to *all* of the files and directories that appear in the trace. TBBT extracts the full pathname of all files and directories through our file system hierarchy extraction algorithm and stores the information in a *hierarchy map*, as described in Section 3.2. The NFS server might use a different *filehandle* for a particular file every time the trace is replayed, but the TBBT-ID will never change.

TBBT also inserts additional information into the trace records to facilitate trace replay. For example, neither a `remove` request nor a `remove` reply contains the *filehandle* of the removed file, which is needed for file system dependency analysis during trace replay (as discussed in Section 3.4.1). The same problem exists for `rmdir` and `rename`. For all three operations, TBBT infers the TBBT-ID of the object in question from the parent’s TBBT-ID, the object name and the file system image, and inserts it into the associated trace record.

Another aspect of TBBT trace rewriting is dealing with errors or omissions in the original trace. The most common error is packet loss. The traces we use for our experiments are reportedly missing as many as 10% of the NFS calls and responses during periods of bursty traffic [7]. The number of lost calls and responses can be estimated by analyzing the progression of RPC exchange IDs (XIDs), which are typically generated by using a simple counter, but this does nothing to tell us *what* was lost.

In many cases, the contents of missing calls may be inferred – although not always with complete certainty. For example, if we observe the request sequence `remove A; remove A` and each of these requests has a successful reply, then it is clear that there must be a `create`, `rename`, `symlink`, or `link` request between the two `remove` requests – some time between when the file “A” is removed the first and second times, another file named “A” must have appeared – but this event is missing from the trace. If we simply replayed the trace without correcting this problem, then then second `remove A` would fail instead of succeed. Such a discrepancy is called a *replay failure*. The correction is to insert some NFS operations to replace the missed packets. Note that it is also a replay failure if the replay of an operation returns success while the original trace recorded failure, or both return failure but with different failures.

We use a table-driven heuristic approach to select corrective operations and insert them into the replay stream. To enumerate all possible combinations of operations, trace return codes, and replay return codes could require

Op	Replay error	Corrective Op(s)
create	file already exists	remove
remove	file does not exist	create
rmdir	directory not empty	remove / rmdir
getattr	permission denied	setattr

Table 2: Examples of trace corrections. In these examples, the operation was observed to succeed in the trace, but would fail during replay. To prevent the failure, corrective operations are added to replay to ensure that the observed operation will succeed.

an enormous table – but in practice, the combinations we have actually encountered all fall into approximately thirty distinct cases. Table 2 illustrates a small number of unexpected replay failures and our resolution rules. Note that there is frequently more than one way to augment the trace in order to prevent the problem. For example, if a file cannot be created because it already exists in the file system, we could either rename the file or remove it. We cannot determine which of these two operations are missing (or whether there are additional operations that we missed as well) but we can observe that removes are almost always more frequent than renames and therefore always choose to correct this problem via a remove.

A similar problem is that we cannot accurately determine the correct timestamp for each corrective operation. Therefore the inserted operations might not perfectly recreate the action of the missing packets. There are also lost packets which do not lead to replay failures and therefore cannot be detected. Since the overall number of lost RPC messages is small (approaching 10% only in extreme situations, and typically about 1%), and most of them are operations which do not require corrective operations such as `readdir`, `read`, `getattr` and `lookup`, the total number of corrective operations is always much smaller (about 0.05%) than the operations taken verbatim from the original trace.

Potential replay failures are detected and corrected through a simulated *pre-play*. The pre-play executes the trace requests one-by-one synchronously. Replay failures are detected by comparing the return value of original request in the trace and the return value of pre-play. The corrective operations are generated according to the trace correction table and are inserted into the trace with a timestamp randomly selected from its valid range.

3.2 Creating the Initial File System Image

To replay calls from a file access trace, the tested server must be initialized with a file system image similar to that of the traced server so that it can respond correctly

to the trace requests. There are two factors to be considered while creating the initial file system image: the logical file system hierarchy and the physical disk layout. While the former is essential for correct trace replay, the latter is crucial to the performance characteristics of the file system. Ideally, one could take a file system snapshot of the traced server before a trace is collected. In practice, however, this is often impractical because it may cause service degradation. Moreover, most file system snapshotting tools capture only the file system hierarchy but not the physical layout. TBBT approximates the traced server’s file system image using information from the NFS trace. It then constructs (and ages) the image through the native file system of the tested server.

The idea of extracting the file system hierarchy from an NFS trace is not new [4, 8]. However, because earlier tools were developed mainly for the purpose of trace studies, the extracted file system hierarchy is not sufficiently complete to permit trace replay. For example, operations such as `symlink`, `link` and `rename` were not handled properly, and the dynamic changes to the file system hierarchy during tracing are not properly captured.

TBBT’s file system hierarchy extraction tool produces a *hierarchy map*. Each entry in the hierarchy map contains a *TBBT-ID*, *path*, *createTime*, *deleteTime*, *size*, and *type*. Each hierarchy map entry corresponds to one file system object under one path. File system objects with multiple hard links have multiple paths and may appear in multiple hierarchy map entries, but have the same TBBT-ID in each entry. If a path exists before trace collection starts, its *createTime* is set to 0 (to indicate that TBBT must create this object before the trace replay begins), and the *size* field gives the object’s size at the time when the trace began. The *type* field indicates whether the file is a regular file, a directory, or a symbolic link.

The file system hierarchy extracted from an NFS trace is not necessarily a complete snapshot of the traced file system because only files that are referenced in the trace appear in the TBBT *hierarchy map* and many workloads are highly localized. In traces gathered from our own systems, we observed that in many cases only a small fraction of a file system is actually accessed during the course of a day (or even a month). The fact that only active files appear in the TBBT *hierarchy map* may have a serious effect on the locality of the resulting file system. To alleviate this problem, TBBT augments the extracted file system hierarchy with additional files. Details about how these objects are created are given in Section 3.3.2.

Given a TBBT hierarchy map, TBBT populates the tested server file system according the order that files appeared in the hierarchy map, creating each file, directory, or link as we encounter it. Usually files in the hierarchy

map are organized in depth-first order, but it could be in other order too, as long as a file appears later than its parent directory. This naive approach yields a nearly ideal physical disk layout for the given file system hierarchy: free space is contiguous, data blocks of each file are allocated together and therefore likely to be physically contiguous, data is close to the corresponding metadata, and files under the same directory are grouped together. As a result, it does not capture the effects of concurrent access and file system aging. TBBT's artificial aging technique is designed to emulate such aging.

3.3 Artificially Aging a File System

The effect of aging centers on fragmented free space, fragmented files, and declustered objects (objects which are often accessed together but are located far from each other on the disk). TBBT's aging mechanism is meant to create such aging effects. The current implementation and evaluation focuses on the fragmentation of file blocks and free space, but the mechanism is extensible to include the declustering effect among related file system objects. TBBT's aging mechanism is purely synthetic and is not meant to emulate the actual file system aging process (as done in Keith Smith's work [20]).

An important design constraint of TBBT's file system aging mechanism is that it should be able to exercise any desired aging effects against any file system without resorting to the raw disk interface. Using only the standard system call interface makes it easier to integrate a file system aging mechanism into other file system benchmarking tools.

Aging is related to the file system block allocation algorithm. Some of our analyses assume a FFS-like block allocation policy. This policy divides a disk partition into multiple *cylinder groups*, each of which has a fixed number of free inodes and free blocks. Files under the same directory are preferentially clustered in one group. Our aging techniques are expected to work well for servers utilizing FFS-like file systems on simple block storage. Other types of systems have not yet been tested.

3.3.1 File System Aging Metrics

To the best of our knowledge, there do not exist any standard metrics to quantify the effect of aging on a file system. Before presenting our file system aging metrics, we define several basic terms used in our discussion.

A file system *object* is a regular file, directory, symbolic link, or a special device. The *free space object* is an abstract object that contains all of the free blocks in the file system. A *fragment* is a contiguous range of blocks within an object. The *fragment size* is the number

of blocks within a fragment, and the *fragment distance* is the number of physical blocks between two adjacent fragments of the same object. The *block distance* is the number of physical blocks between two adjacent logical blocks in an object. The *inode distance* is the number of physical blocks between an object's inode and its first data block and the *parent distance* is the number of physical blocks between the first block of an object and that of its parent directory. The block used in these definitions is file system block (4KB by default).

If we assume that the goal of the policy used to allocate blocks for a file is to allocate them sequentially, then the effect of file system aging (in terms of the fragmentation it causes) can be quantified in terms of the physical distance between consecutive blocks of a file. *Average fragment distance*, *average block distance* and *average fragment size* are calculated over all fragments/blocks that belong to each file within a file system partition, and are related to one another as follows: $average\ fragment\ distance = average\ block\ distance \times average\ fragment\ size$. Because the calculation of these metrics is averaged over the number of blocks or segments in a file, files of different size are weighted accordingly. *Average block distance* describes the overall degree of file fragmentation. Either of the other two metrics helps further distinguish between the following two types of fragmentation: a large number of small fragments that are located relatively close to each other, or a small number of large fragments that are located far away from each other. *Average inode distance* can be considered as a special case of *average block distance* because it measures the distance between a file's inode and its first data block.

In an aged file system, both free space and allocated space are fragmented. The *average fragment size* of the special *free space object* reflects how fragmented the free space portion of a file partition is. The file system aging effect can also be quantified based on the degree of clustering among related files, e.g., files within the same directory. The *average parent distance* is meant to capture the proximity of a directory and the files it contains, and indirectly the proximity of files within the same directory. Alternatively, one can compute *average sibling distance* between each pair of files within the same directory.

These metrics provide a simplistic model; they do not capture the fact that logical block distances do not equate to physical seek time nor do they reflect the non-commutative nature of rotational delays, which make it common for disk head to take a different amount of time to move from position A to position B than from B to A. This simplistic model does have several benefits, however: it is both device and file-system independent, and does provide intuition for the file system performance.

3.3.2 File System Aging Techniques

Free space fragmentation is due primarily to file deletions. Fragmented files, in contrast, are caused by two reasons. First, a file will become fragmented when free space is fragmented and there are no contiguous free blocks to allocate when the file grows. Second, interleaving of append operations to several files may cause blocks associated with these different files to be interleaved as well. There are several techniques that mitigate the fragmentation effect of interleaved appends, such as dividing a logical partition into *cylinder groups* and placing files in different cylinder groups [14]. Another heuristic is to preallocate contiguous blocks when a file is opened for writing. Despite these optimizations, files can still get fragmented if interleaved appends occur within the same group or if the file size is more than the pre-allocated size.

The aging effects become more pronounced when inode and block utilization is unbalanced between cylinder groups. To reduce the declustering effect, an FFS-like policy tries to place files under the same directory in one group, and to allocate one file's inode and its data blocks in the same group. But it also tries to keep balanced utilizations among different *cylinder groups*. Once the utilization of a group is too high, allocation switches to another group unless there are no available cylinder groups. The unbalanced usage is usually caused by a highly skewed directory tree where some directory has many small files or some directory has very large files.

TBBT uses interleaved appending as the primary file system aging technique, and uses file deletion only to fragment the free space. Given a file system partition, TBBT's initialization procedure populates it with the initial file system hierarchy derived from the input trace and additional synthetic objects to fill all available space. These synthetic objects are used both to populate the incomplete file system hierarchy and to occupy free space. All of the objects become fragmented because of interleaved appending. At the end of the initialization, the synthetic objects that occupy the free space are deleted to have fragmented free space available. This way, to initialize a 1GB file system partition with 0.1GB free space, we write exactly 1GB of data, then we delete 0.1GB of data. In contrast, Smith's aging technique writes around 80GB of data, and deletes around 79GB of data. Note that our choice of terminology and examples in this discussion assumes that the underlying file system uses an FFS-like strategy for block allocation. We believe that our methodology works just as well with other strategies, such as LFS, although for LFS instead of fragmenting the free space, we are creating dead blocks for the cleaner to find and reorganize.

To determine the set of synthetic objects to be added to a file system and generate a complete TBBT hierarchy map, TBBT takes four parameters. The first two parameters are *file size distribution* and *directory/file ratio*, which are similar to SPECsfs's file system initialization parameters. The third parameter is the *distortion factor*, which determines the degree of imbalance among directories in terms of directory fan-out and the file size distribution within each directory. The fourth parameter is the *merge factor*, which specifies how extensively synthetic objects are commingled with the initial file system image. A low *merge factor* means that most directories are dominated by either synthetic or extracted objects, but not both.

To create fragmentation, TBBT interleaves the append operations to a set of files, and in each append operation adds blocks to the associated file. To counter the file pre-allocation optimization technique, each append operation is performed in a separate open-close session. File blocks written in an append operation are likely to reside in contiguous disk blocks. However, blocks that are written in one append operation to a file may be far away from those blocks that are written in another append operation to the same file. The expected distance between consecutive fragments of the same file increases with the total size of files that are appended concurrently. By controlling the total size of files involved in interleaved appending, called **interleaving scope**, and the number of blocks in each append operation, TBBT can control the *average block distance* and *average fragment size* of the resulting file system. We assume that large files tend to be written in larger chunks. Instead of directly using the number of blocks in each appending operation to tune *average fragment size*, we use a parameter called **append operations per file**, which specifies the number of appending operations used to initialize one file. The minimum size of each fragment is 1 block. Usually the average file size is around 10 blocks. Therefore, a very large value of *append operations per file* may only affect some large files.

The declustering effect is described by *average inode distance* and *average parent/sibling distance*. To create the declustering effect, TBBT may add zero-sized synthetic objects to create a skewed directory hierarchy and provoke unbalanced usage among cylinder groups. To increase the *average parent/sibling distance*, rather than picking files at random, TBBT interleaves files from different directories.

In summary, given a TBBT hierarchy map, TBBT's file system aging mechanism tries to tune: *average block distance* and *average fragment size* of normal file, *average fragment size* of the special *free space object*, *average inode distance* and *average parent/sibling dis-*

tance. *Average block distance* is tuned via the *inter-leaving scope*. *Average fragment size* is tuned via the *append operations per file*. Moreover, different aging effects could be specified for different files, including the special *free space object*. We have not implemented the controls for *average inode distance* and *average parent/sibling distance* in the current TBBT prototype. Randomization is used to avoid regular patterns. TBBT's aging technique can be used to initialize the file system image for both trace-based and synthetic workload-based benchmarking.

3.4 Trace Replay

When replaying the requests in an input trace, TBBT needs to respect the semantics of the NFS protocol. Sending out requests according to their timestamps is not always feasible. For example, given sequence1 in Table 3, if the create reply comes at time 3 during the replay, it is impossible to send the write request at time 2. TBBT's trace player provides flexible policies to handle the issue of when it is correct to send a request. For SPECsfs-like synthetic benchmarks, multiple processes are used to generate requests against multiple disjoint directories, and in each process requests are executed synchronously without any concurrency. As a result, the SPECsfs load generation policy is much simpler.

3.4.1 Ordering and Timing Policy

TBBT's trace player provides two ordering policies to determine the relative order among requests: *conservative order* and *FS dependency order*. Both guarantee the replay can proceed to the end, and both result in same modifications to the initial file system hierarchy at the end of trace play. TBBT also provides two timing policies: *full speed* and *timestamp-based*, to determine the exact time at which requests are issued. In the *full speed* policy, requests are dispatched as quickly as possible, as long as the chosen ordering policy is obeyed. In the *timestamp based* policy, requests are dispatched as close to their timestamp as possible without violating the ordering policy.

When the *conservative order* policy is used, a request is issued only after all prior requests (e.g., requests with earlier timestamps) have been issued and all prior replies have been received. The *conservative order* captures some of the concurrency inherent in the trace although it will not generate a workload with higher concurrency. In contrast, there is no concurrency in the workload generated by each process of SPECsfs's load generator.

Because of differences in the traced server and tested server, it is impossible to guarantee that the order of

T	sequence1	sequence2	sequence3	sequence4
0	create A call	create A call	create A call	create A call
1	create A reply	create A reply		write B call
2	write A call	write B call		write B reply
3	write A reply	write B reply	create A reply	create A reply
4			write B call	
5			write B reply	

Table 3: Examples to illustrate the ordering issue in trace replay. The first column represent normalized time. Other columns represent NFS request sequence examples. The create latency is 1 on the traced server and 3 on the tested server. In Sequence1 there is FS-level dependency because both operations involves the same file. In Sequence2 there is no FS-level dependency but may an have application-level dependency. Sequence3 is the result of replaying sequence2 by *conservative order*. Sequence4 is the result of playing sequence2 by *FS dependency order*.

replies in the trace replay is exactly the same as that in the trace. The disadvantage of *conservative order* is that processing latency variations in the tested server may unnecessarily affect its throughput. For example, given sequence2 of Table 3, if the create latency during replay is three times higher than the latency in the original trace, the request issue ordering becomes sequence3 which has a lower throughput than sequence2.

In the *FS dependency order* policy, the dependencies of each request on other earlier requests and replies in the trace are discovered via a read/write serialization algorithm. With the *FS dependency order* policy, the request issue ordering for sequence2 in Table 3 becomes sequence4, which results in higher throughput than sequence3. Conceptually, the file system hierarchy is viewed as a shared data structure and each NFS request is a read or write operation on one or more parts of this structure. If an NFS operation modifies some part of the structure that is accessed by a later operation in the trace, then the latter operation cannot be started until the first has finished. For example, it is dangerous to overlap a request to create a file and a request to write some data to that file; if the write request arrives too soon, it may fail because the file does not yet exist. In many cases it is not necessary to wait for the response, but simply to make sure that the requests are made in the correct order. The exceptions are the replies from `create`, `mkdir`, `symlink`, and `mknod`. These replies are regarded as a write operation to the newly created object, and therefore need to be properly serialized with respect to subsequent accesses to these newly created objects. Table 4 summarizes the file system objects that are read or written by each type of request and re-

ply. Because concurrent access to the same file system object is infrequent in real NFS traces, the granularity of TBBT's dependency analysis is an individual file system object. For finer-granularity dependency analysis, inode attributes and each file block could be considered separately.

The *FS dependency order* may be too aggressive because it only captures the dependencies detectable through the shared file system data structure but does not discover application-level dependencies. For example, in Table 3, if the application logic is to write some debug information to the log file B after each successful create A operation, then the write operation indeed depends on the create operation and should be sent after receiving the create request's reply. In this case, ordering requests based FS-level dependencies is not sufficient. In general, *conservative order* should be used when *FS dependency order* cannot properly account for many application-level dependencies.

3.4.2 Workload Scaling

Given a trace, TBBT can scale it up or down spatially or temporally. To spatially scale up a trace, the trace and its initial file system image are cloned several times, and each cloned trace is replayed against a separate copy of the initial image. Spatial scale-up is analogous to the way that synthetic benchmarks run multiple load-generation processes. To spatially scale down a trace, the trace is decomposed into multiple sub-traces, where each sub-trace accesses only a proper subset of the initial file system image. Not all traces can be easily decomposed into such sub-traces, but it is typically not a problem for traces collected from file servers that support a large number of clients and users. Users don't often share any files and we can just take a subset of them.

Temporally scaling up or down a trace is implemented by issuing the requests in the trace according to the scaled timestamp, while observing the chosen ordering policy. An ordering policy from above bounds the temporal scaling of a given trace. The two scaling approaches can be combined to scale a trace. For example, if the required speed-up factor is 12, it can be achieved by a spatial scale-up of 4 and a temporal scale-up of 3.

4 Implementation

Trace transformation and initial file hierarchy extraction are implemented in Perl. Trace replay is implemented in C. Each trace is processed in three passes. The first pass transforms the collected trace into TBBT's trace format, except the TBBT-ID field in the replies to `remove`,

`rmdir`, and `rename`. The second pass corrects trace errors by a pre-play of the trace. The third pass extracts the *hierarchy map* and adds the TBBT-ID to the replies to `remove`, `rmdir`, `rename`. Each successful or failed directory operation may contain information about a <parent, child> relationship from which the *hierarchy map* is built. Hierarchy extraction consumes a great deal of CPU and memory, especially for traces of large size. An incremental version of hierarchy extraction is possible and will greatly improve its efficiency.

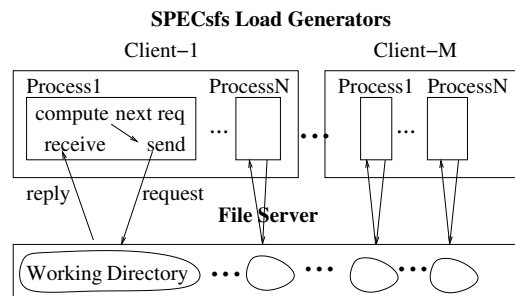


Figure 1: SPECSfs uses multiple independent processes to generate requests targeted at disjoint directories.

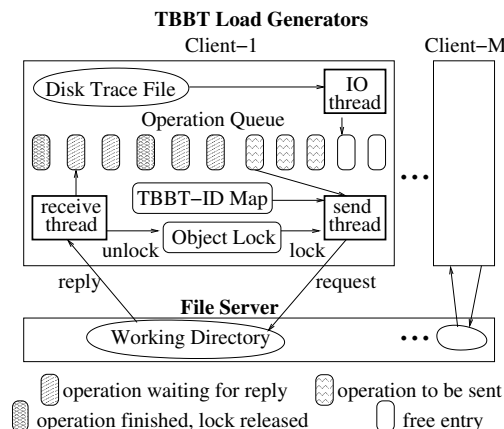


Figure 2: TBBT uses a three-thread process to read and replay traces stored on disk.

Similar to SPECSfs [21], TBBT's trace player bypasses the NFS client and sends NFS requests directly to the tested file server using user-level RPC. The software architecture of the TBBT player, however, is different from SPECSfs. As shown in Figure 1, the workload generator of SPECSfs on each client machine uses a multi-process software architecture, with each process dispatching NFS requests using synchronous RPC. In contrast, TBBT uses a 3-thread software structure, as shown in Figure 2, which is more efficient because it reduces the context switching and scheduling overhead. The *I/O thread* continuously reads trace records into a

request/reply	shared-data-structure set	type
REQ: read/readdir/getattr/readlink obj	obj	'read'
REQ: write/setattr/commit obj	obj	'write'
REQ: lookup parent, name([obj])	parent, [obj]	'read'
REQ: create/mkdir parent, name	parent	'write'
REPLY: create/mkdir obj	obj	'write'
REQ: remove/rmdir parent, name([obj])	parent, [obj]	'write'
REQ: symlink parent, name, path	parent	'write'
REPLY: symlink [obj]	[obj]	'write'
REQ: rename parent1, name1, parent2, name2([obj2])	parent1, parent2, [obj2]	'write'
all other replies	empty	-

Table 4: The file system objects that are read or written by different requests and replies. The notation [obj] means that the object may not exist and therefore the associated operation might return a failure.

cyclic memory buffer called the *operation queue*. The *send thread* and *receive thread* send NFS requests to and receive their replies from the NFS server under test using asynchronous RPC requests. The *operation queue* is also called the *lookahead window*. The size of the *lookahead window* should be several times larger than the theoretical concurrency bound of the input trace to ensure that the *send thread* is always able to find enough concurrent requests at run time.

The *send thread* determines whether an NFS request in the input trace is ready to be dispatched by checking (1) whether it follows the ordering policy (2) whether the request's timestamp is larger than the current timestamp, and (3) whether the number of outstanding requests to a given server exceeds the given threshold. The second check is only for *timestamp-based* policy. The third is to avoid overloading the test file server. The first check is straightforward in the case of *conservative order*. For *FS dependency order*, we use *object locking* as illustrated in Figure 2. Before dispatching an NFS request, the *send thread* acquires the read/write lock(s) on all the object(s) associated with the request (and the reply for operations that refer to additional objects in their reply). Some locks are released after the request is dispatched, other locks are released after the reply is received.

During trace replay, requests are pre-determined, rather than computed on the fly according to current replay status, as in some synthetic benchmarks. This means that a robust trace player needs to properly react to transient server errors or failures in such a way that it can continue to play the trace for as long as possible. This requires the trace player to identify subsequent requests in the trace that are affected by a failed request, directly or indirectly, and skip them, and to handle various run-time errors such that their side effects are effectively contained. For example, because a `create` request is

important for a trace replay to continue, it will be retried multiple times if the request fails; however, a failed read request will not be retried so as not to disrupt the trace replay process.

5 Evaluation

In this section, we examine the validity of the trace-based file system benchmarking methodology, analyze to what extent we may scale the workload, explore the difference between the evaluation results from TBBT and SPECsfs and conclude with a measure of the run-time cost of our TBBT prototype.

The NFS traces used in this study were collected from the EECS NFS server (EECS) and the central computing facility (CAMPUS) at Harvard over a period of two months in 2001 [7]. The EECS workload is dominated by metadata requests and has a read/write ratio of less than 1.0. The CAMPUS workload is almost entirely email and is dominated by reads. The EECS trace and the CAMPUS trace grow by 2 GBytes and 8 GBytes per day, respectively. Most of the Harvard traces have a packet loss ratio of between 0.1-10%.

We use TBBT to drive two NFS servers. The first is the Linux NFSv3 and the second is a repairable file system called RFS. RFS augments a generic NFS server with fast repairability without modifying the NFS protocol or the network file access path [24]. The same machine configuration is used for post-collection trace processing, hosting the test file systems, and running TBBT trace player and SPECsfs benchmark. The machine has a 1.5-GHz Pentium 4 CPU, 512-MByte of memory, and one 40-GByte ST340016A ATA disk drive with 2MB on-disk cache. The operating system is RedHat 7.1.2 with Linux kernel 2.4.7.

5.1 Validity of Trace-Based Evaluation

An ideal trace analysis and replay tool should be able to faithfully recreate the initial file system image and its disk layout, and replay back the requests in the trace with accurate timing. In this section, we evaluate how successful TBBT is in approximating this ideal. Because the Buttress [2] project has already solved the trace replay timing problem, this issue is omitted here.

5.1.1 Extraction of File System Hierarchy

We measured the number of disjoint directory subtrees, the number of directories, the number of files, and the total file system size of the derived file system hierarchy. Figure 3 shows the results for the EECS trace from 10/15/2001 to 10/29/2001. The Y-axis is in logarithmic scale. The total file system size on the EECS server is 400 GB, but only 42 GB is revealed by this 14-day trace (and most of this is discovered during the first several days). We expect the rate will further slow if additional weeks are added. Part of the reason that the extracted hierarchy is incomplete is because this file server has 2 network access interfaces and the NFS traces are collected from only one of the interface. Another reason is that the backup traffic is filtered out of the traces. If they had not been filtered out, then capturing a full backup would give an excellent picture. In general, our results indicate that when the initial file system hierarchy is not available, the hierarchy extracted from the trace may be only a small fraction of the real hierarchy and therefore it is essential to introduce artificial file objects in order to have comparable disk layout.

5.1.2 Effectiveness of Artificial Aging Techniques

In the following experiments, both file system and disk prefetch are enabled, and the file system aging metrics are calculated using disk layout information obtained from the *debugfs* utility available for the ext2 and ext3 file system. We applied our aging technique to two test file systems. The first is a researcher's home directory (which has been in continuous use for more than 1.5 years) and the second is the initial file system image generated by the SPECsfs benchmark.

From the researcher home directory, we selected two subdirectories, *dir1* and *dir2*, and for each subdirectory, created three different versions of the disk image. The first version is a *naturally aged* version, which is obtained by copying the logical partition which contains the original subdirectory to another logical partition using *dd*. The second version is a *synthetically aged* version, which is created through our aging techniques. The

third version represents a *linearized* version of the original subdirectory's disk image using *cp -r* and thus also corresponds to a near-optimal disk image without aging effect. The file system buffer cache is purged before each test. For each of the three versions of each subdirectory, we measure the elapsed time of the command *grep -r*, which is a disk-intensive command that typically spends at least 90% of its execution time waiting for the disk. Therefore, aging is expected to have a direct effect on the execution time of the *grep* command.

Figures 4 and 5 show that the proposed aging technique has the anticipated impact on the performance of *grep* for *dir1* and *dir2*: more interleaving and finer-grained appends result in more fragmentation in the disk image, which leads to lower performance.

Moreover, with the proper aging parameters, it is actually possible to produce a synthetically aged file system whose *grep* performance is the same as that of the original naturally aged file system. For Figure 4, the *<interleaving scope, append operations per file>* pairs that correspond to these cross-over points are *<2,8>*, *<4,2>*, and *<16,1>*. For Figure 5, the *<interleaving scope, append operations per file>* pairs that correspond to these cross-over points are *<4,16>*, *<16,8>*, *<64,4>*, *<256,2>*, and *<4096,1>*. These results demonstrate that the proposed aging technique can indeed produce a realistically aged file system image. However, the question of how to determine aging parameters automatically remains open.

Figures 4 and 5 also show that the *grep* performance of the original naturally-aged image is not very different from that of the linearized image; the impact of natural aging is not more than 20%. TBBT's aging technique can generate much more dramatic aging effects, but it is not clear whether such aging occurs in practice.

To show that the proposed aging technique can be used together with a synthetic benchmark such as SPECsfs, we run the SPECsfs benchmark on the image initialized by SPECsfs itself and the image initialized by the aging technique with different parameters. We used the *append operations per file* value of 4 and varied the *interleaving scope* value, and measured the average read latency and the initial image creation time. The results are shown in Table 5. As expected, the average read latency increases as the initial file system image is aged more drastically. The first two rows show that the SPECsfs benchmark run itself increases the average block distance too and the increase is more observable when the initial image is less aged (first two columns). Finally, the time required to create an initial file system image in general increases with the degree of aging introduced. From the table, TBBT initialization is faster than SPECsfs. The reason could be that SPECsfs uses concurrent initialization pro-

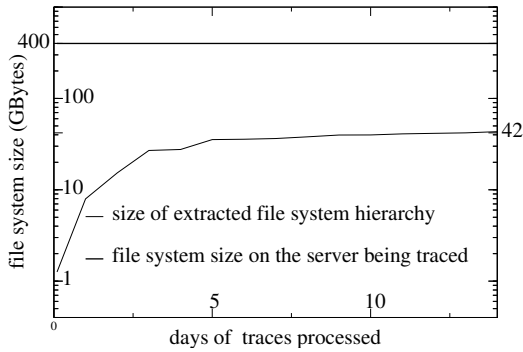


Figure 3: The file system hierarchy size discovered over time. Longer traces provide more information about the file system hierarchy, but with rapidly diminishing returns.

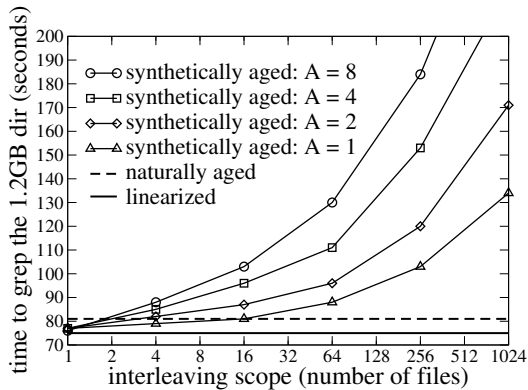


Figure 4: The elapsed time to complete `grep -r` on `dir1` consistently increases with the *interleaving scope* and the *append operations per file* parameter. Different curves corresponding to different values of the *append operations per file* parameter. For example, “A = 8” means the *append operations per file* is 8. The lines for the *naturally aged* and *linearized* image are flat because no synthetic aging is applied to these two cases.

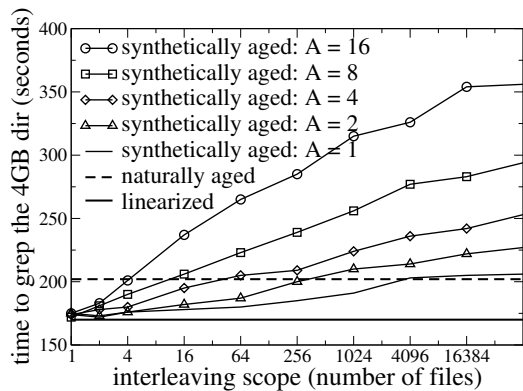


Figure 5: This figure shows the result of conducting a similar experiment as Figure 4 on a much larger research project directory. The two figures show that the aging parameters create qualitatively similar but quantitatively different aging effects on different file system data.

cesses and the overall disk access pattern does not have as good locality as the single-threaded TBBT initialization procedure.

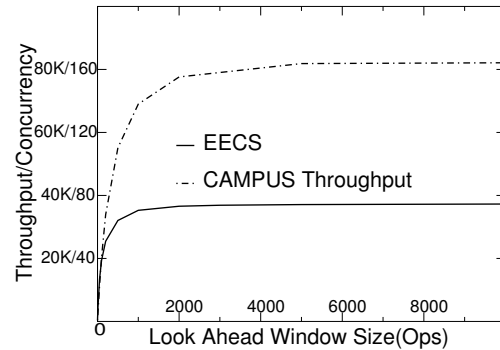


Figure 6: The impact of the *lookahead window* size on the concurrency and thus the throughput of the workload that TBBT can generate from the EECS and CAMPUS trace.

5.2 Workload Scaling

To study the maximum concurrency available in a trace, we conducted a simulation study. The simulation assumes the reply for each request always comes back successfully after a pre-configured latency. The throughput result is given in Figure 6. In this simulation, there are two factors that limit the maximum concurrency: the *lookahead window* in which future requests are examined during the simulation, and the per-request latency at the server. For per-request latency, we used the latency numbers in Table 6. Figure 6 shows the correlation between the maximum throughput that can be generated and the lookahead window size.

The simulation results show that even for a lightly loaded workload such as the EECS trace (30 requests/sec) and a modest *lookahead window* size (4000), there is enough concurrency to drive a file server with a performance target of 37000 requests/sec using temporal scaling.

5.3 Comparison of Evaluation Results

We conducted experiments to evaluate two file servers: NFS and RFS using both TBBT and the synthetic benchmark SPECsfs. In these experiments, the tested file system is properly warmed up before performance measurements are taken. We first played the EECS trace of 10/21/2001, and tried to tune the parameters of the SPECsfs benchmark so that they match the trace’s characteristics as closely as possible. We also changed the source code of SPECsfs so that its file size distribution

	SPECsfs initialization	scope=512	scope=8192	scope=65536
average block distance before SPECsfs run	2	20	2180	6230
average block distance after SPECsfs run	817	828	2641	6510
average read latency	3.24 msec	3.24 msec	3.31 msec	4.45 msec
time to create initial image	683 sec	330 sec	574 sec	668 sec

Table 5: Results of applying the proposed file system aging techniques to SPECsfs. First column gives result of using SPECsfs’s own initialization procedure. Other three columns show the result of using TBBT’s aging technique to create SPECsfs run’s initial file system image.

NFS 10/21/01	original		scale-up		peak load	
benchmark	S	T	S	T	S	T
throughput (Ops)	33	30	189	180	1231	1807
getattr (msecs)	5.1	0.6	0.9	1.5	2.1	0.7
lookup (msecs)	2.9	0.9	0.8	2.0	2.0	1.2
read (msecs)	9.6	3.1	5.3	4.8	5.4	4.7
write (msecs)	9.7	2.2	4.4	3.8	4.6	2.5
create (msecs)	0.5	0.7	0.7	0.9	17.3	0.7

Table 6: Per-operation latency and overall throughput(operations per second) comparison between TBBT and SPECsfs for an NFS server using the EECS 10/21/2001 trace. “T” means TBBT, “S” means SPECsfs.

RFS 10/21/01	original		scale-up		peak load	
benchmark	S	T	S	T	S	T
throughput (Ops)	32	30	187	180	619	1395
getattr (msecs)	4.0	0.7	2.2	1.2	3.2	0.8
lookup (msecs)	4.4	0.7	2.8	1.3	2.6	1.0
read (msecs)	10.8	3.3	8.4	4.1	18.1	4.9
write (msecs)	11.6	5.4	7.4	4.0	11.1	2.8
create (msecs)	0.7	1.0	5.1	1.3	16.3	1.2

Table 7: Performance results for RFS server using the EECS 10/21/2001 trace.

matches the file size distribution in the 10/21/2001 trace. The maximum throughput of the Linux NFS server under SPECsfs is 1231 requests/sec, and is 1807 requests/sec under TBBT. The difference is a non-trivial 46.8%. In terms of per-operation latency, Table 6 shows the latency of five different operations under the original load (30 requests/sec), under a temporally scaled load with a speed-up factor of 6, and under the peak load. The per-operation latency numbers for TBBT and for SPECsfs are qualitatively different in most cases.

The same experiment, using RFS instead of the Linux NFS server, is shown in Table 7. The maximum throughput is 619 requests/sec for SPECsfs versus 1395 requests/sec for TBBT – a difference of 125.4%. Again there is no obvious relationship between the average per-operation latency for SPECsfs and TBBT.

NFS 10/22/01	original		scale-up		peak load	
benchmark	S	T	S	T	S	T
throughput (Ops)	16	15	191	187	2596	4125
getattr (msecs)	4.7	0.5	0.7	0.7	1.02	0.7
lookup (msecs)	2.8	0.6	0.5	0.8	1.01	0.6
read (msecs)	10.3	2.1	19.7	3.1	7.4	4.2
write (msecs)	7	1.0	6.3	1.2	3.8	3.0
create (msecs)	0.5	0.9	1.2	0.5	7.9	0.7

Table 8: Performance results for NFS server using the EECS 10/22/2001 trace.

To determine whether these differences are consistent across traces taken from different days, we ran the 10/22/2001 EECS trace against the LINUX NFS server. The 10/22/2001 trace is dominated by metadata operation (80%) while the 10/21/2001 trace has substantial read/write operations (60%). The SPECsfs configuration is again tuned to match the access characteristics of the 10/22/2001 trace. The results in Table 8 show that the difference between TBBT and SPECsfs in throughput and per-operation latency is still quite noticeable.

5.4 Implementation Efficiency

TBBT’s post-collection trace processing procedure can process 2.5 MBytes of trace or 5000 requests per second. TBBT’s initialization time increases with the total file system size as well as the degree of file system aging desired, because the more drastic the aging effect TBBT attempts, the less is the disk access locality in its file system population process. Table 5 shows TBBT’s initialization time is also affected by and *average block distance level*. Overall TBBT’s aging techniques is very efficient. The initialization speed is more than two orders of magnitude faster than Smith’s aging technique.

The CPU load of TBBT comes from the send thread, receive thread, and the network subsystem inside the OS. When the Linux NFS server runs under a trace at peak throughput (1807 requests/sec), the measured CPU utilization and network bandwidth consumption for TBBT’s trace player are 15% and 60.5 Mbps. When the

same Linux NFS server runs under a SPECsfs benchmark at peak throughput (1231 requests/sec), the measured CPU utilization and network bandwidth consumption for the SPECsfs workload generator are 11% and 37.9 Mbps. These results suggest that TBBT's trace player is actually more efficient than SPECsfs's workload generator (in terms of CPU utilization per NFS operation) despite the fact that TBBT requires additional disk I/O for trace reads, and incurs additional CPU overhead for dependency detection and error handling. We believe that part of the reason is because TBBT uses only three threads, whereas SPECsfs uses multiple processes.

6 Limitations

There are several limitations associated with the proposed trace-driven approach to file system evaluation. First, for a given input workload, TBBT assumes the trace gathered from one file system is similar to that from the file system under test. Unfortunately, this assumption does not always hold, because even under the same client workload, it is possible that different file servers based on the same protocol produce very different traces. For example, file mount parameters such as read/write/readdir transfer sizes could have a substantial impact on the actual requests seen by an NFS server.

Second, there is no guarantee that the heuristics used to scale up a trace are correct in practice. For example, if the bottleneck of a trace lies in accesses to a single file or directory, then cloning these accesses when replaying the trace is not feasible.

Third, it is generally not possible to deduce the entire file system hierarchy or its on-disk layout by passive tracing. Therefore the best one can do is to estimate the size distribution of those files that are never accessed during the tracing period and to apply synthetic aging techniques to derive a more realistic initial file system image. Again the file aging techniques proposed in this paper are not meant to reproduce the actual aging characteristics of the trace's target file system, but to provide users the flexibility of incorporate some file aging effects in their evaluations.

Fourth, trace-based evaluation is not as flexible as those based on synthetic benchmarks, in terms of the ability to explore the entire workload space. Consequently, TBBT should be used to complement synthetic benchmarks rather than replace them.

Finally, TBBT replays write request with synthetic data blocks. This has no effect on a NFS server built on top of a conventional file system, but is not correct for storage systems whose behavior depends on the data being written (i.e., content-addressed storage systems).

7 Conclusion

The prevailing practice of evaluating the performance of a file system/server is based on synthetic benchmarks. Modern synthetic benchmarks do incorporate important characteristics of real file access traces and are capable of generating file access workloads that are representative of their target operating environments. However, they rarely fully capture the time-varying and oftentimes subtle characteristics of a specific site's workload. In this paper, we advocate a complementary trace-driven file system evaluation methodology, in which one evaluates the performance of a file system/server on a site by driving it with file access traces collected from that site. To support this methodology, we present TBBT, the first comprehensive NFS trace analysis and replay tool. TBBT is a turn-key system that can take an NFS trace, properly initialize the target file server, drive it with a scaled version of the trace, and report latency and throughput numbers. TBBT addresses most, if not all, of the trace-driven workload generation problems, including correcting tracing errors, automatic derivation of initial file system from a trace, aging the file system to a configurable extent, preserving the dependencies among trace requests during replay, scaling a trace to a replay rate that can be higher or lower than the speed at which the trace is collected, and graceful handling of trace collection errors and implementation bugs in the test file system/server. Finally, we show that all these features can be implemented efficiently such that a single trace replay machine can stress a file server with state-of-the-art performance.

In addition to being a useful tool for file system researchers, perhaps the most promising application of TBBT is to use it as a site-specific benchmarking tool for comparing competing file servers using the same protocol. That is, one can compare two or more file servers for a particular site by first collecting traces on the site, and then testing the performance of each of the file servers using the collected traces. Assuming traces collected on a site are indeed representative of that site's workload, comparing file servers using such a procedure may well be the best possible approach. TBBT is available at http://www.ecsl.cs.sunysb.edu/TBBT/TBBT_dist.tgz.

Acknowledgments

We thank Daniel Ellard of Sun Microsystems Laboratories for his contributions to the TBBT design discussions, his NFS traces and analysis tools, and his help with early drafts of this paper. We thank our shepherd, Rod Van Meter, for his critical opin-

ions, detailed suggestions and great patience. We thank the anonymous reviewers for their valuable comments. We thank our ECSL colleagues for their help in paper writing and testbed setup. This research is supported by NSF awards ACI-0234281, CCF-0342556, SCI-0401777, CNS-0410694 and CNS-0435373 as well as fundings from Computer Associates Inc., New York State Center of Advanced Technology in Sensors, National Institute of Standards and Technologies, Siemens, and Rether Networks Inc.

References

- [1] Darrell Anderson and Jeff Chase. Fstress: A Flexible Network File Service Benchmark. Technical Report TR-2001-2002, Duke University, May 2002.
- [2] Eric Anderson. A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST'04)*, March 2004.
- [3] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212, Monterey, CA, October 1991.
- [4] Matthew A. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, San Francisco, CA, January 1992.
- [5] Aaron Brown and Margo Seltzer. Operating system benchmarking in the wake of Imbench: A case study of the performance of NetBSD on the Intel x86 architecture. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'97)*, 1997.
- [6] Michael D. Dahlin. A quantitative analysis of cache policies for scalable network file systems. In *First OSDI*, pages 267–280, 1994.
- [7] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.
- [8] Daniel Ellard and Margo Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Seventeenth Annual Large Installation System Administration Conference (LISA'03)*, pages 73–85, San Diego, CA, October 2003.
- [9] Steven. L. Gaede. *Perspectives on the SPEC SDET benchmark*. Lone Eagle Systems Inc., January 1999.
- [10] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–284. ACM Press, 1997.
- [11] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and Performance in a Distributed System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [12] Jeffrey. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance, October 1997.
- [13] Richard McDougall. A new methodology for characterizing file system performance using a hybrid of analytic models and a synthetic benchmark. In *Work in Progress Session in 3rd Usenix Conference on File and Storage Technologies*, March 2004.
- [14] Marshall K. McKusick, William N. Joy, Samuel J. Lefler, and Robert S. Fabry. A Fast File System for UNIX. *Computer Systems*, 2(3):181–197, 1984.
- [15] Larry W. McVoy and Carl Staelin. Imbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, pages 279–294, 1996.
- [16] Michael Mesnier, Eno Thereska, Daniel Ellard, Gregory R. Ganger, and Margo Seltzer. File classification in self-* storage systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC)*, pages 44–51, May 2004.
- [17] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP'85)*, pages 15–24, Orcas Island, WA, December 1985.
- [18] Drew Roselli, Jacob Lorch, and Thomas Anderson. A comparison of file system workloads. In *USENIX 2000 Technical Conference*, pages 41–54, San Diego, CA, 2000.
- [19] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX Annual Technical Conference*, 2000.
- [20] Keith A. Smith and Margo I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of SIGMETRICS 1997: Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.
- [21] SPEC SFS (System File Server) Benchmark, 1997. <http://www.spec.org/osg/sfs97r1/>.
- [22] Werner Vogels. File System Usage in Windows NT. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 93–109, Kiawah Island, SC, December 1999.
- [23] Barry L. Wolman and Thomas M. Olson. IOBENCH: a system independent IO benchmark. *Computer Architecture News*, 17(5):55–70, September 1989.
- [24] Ningning Zhu and Tzi-cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, June 2003.

Accurate and Efficient Replaying of File System Traces

Nikolai Joukov, Timothy Wong, and Erez Zadok

Stony Brook University

{kolya,tim,ezk}@fsl.cs.stonybrook.edu

Abstract

Replaying traces is a time-honored method for benchmarking, stress-testing, and debugging systems—and more recently—forensic analysis. One benefit to replaying traces is the reproducibility of the exact set of operations that were captured during a specific workload. Existing trace capture and replay systems operate at different levels: network packets, disk device drivers, network file systems, or system calls. System call replayers miss memory-mapped operations and cannot replay I/O-intensive workloads at original speeds. Traces captured at other levels miss vital information that is available only at the file system level.

We designed and implemented *Replayfs*, the first system for replaying file system traces at the VFS level. The VFS is the most appropriate level for replaying file system traces because all operations are reproduced in a manner that is most relevant to file-system developers. Thanks to the uniform VFS API, traces can be replayed transparently onto any existing file system, even a different one than the one originally traced, without modifying existing file systems. *Replayfs*'s user-level compiler prepares a trace to be replayed efficiently in the kernel where multiple kernel threads prefetch and schedule the replay of file system operations precisely and efficiently. These techniques allow us to replay I/O-intensive traces at different speeds, and even accelerate them on the same hardware that the trace was captured on originally.

1 Introduction

Trace replaying is useful for file system benchmarking, stress-testing, debugging, and forensics. It is also a reproducible way to apply real-life workloads to file systems. Another advantage of traces is their ease of distribution.

For *benchmarking*, synthetically generated workloads rarely represent real workloads [42]. Compile benchmarks [14] put little load on the file system and are not scalable [6]. Real workloads are more complicated than artificially created ones, so traces of real file system activity make for better test workloads. They represent the actual file system workloads and can be scaled as needed. In addition to scaling, captured traces can be modified in many ways before being replayed. For example, modification of the disk block locations can help to evaluate new disk layout policies [27]. Sometimes trace replaying is used to test a target file system with a synthetic

workload for which application reproducibility is difficult. One of the most common examples is the replaying of TPC [35] traces on file systems, because running the actual benchmark is complicated and requires a database system. Also, replaying traces that were captured by others allows a fair comparison of file systems. Finally, trace replaying is an accurate method to prepare a file system for benchmarking by aging it [31].

Synthetic benchmarks may be more predictable than real-life workloads, but they do not exercise many possible file system operation sequences. Replaying can be used to *stress test* a file system under practical conditions.

Trace replaying allows for selectively replaying portions of a trace. This is useful to narrow down the search for problems during *debugging*. Precise timing and minimal influence on the system being tested are key requirements to reproduce the exact timing conditions.

Replaying file system traces can be considered a form of fine-grained versioning. Existing versioning file systems [20, 32] cannot reproduce the timing and in-memory conditions related to file system modifications. This makes trace replaying a better choice for *forensic* purposes. Replaying traces back and forth in time can be useful for post-mortem analysis of an attack.

File system traces can be captured and replayed at different logical levels: system calls, the Virtual File System (VFS), the network level for network file systems, and the driver level. The easiest way to collect and replay file system traces is by recording and reissuing system calls entirely from user mode. Unfortunately, this method does not capture memory-mapped file system operations. This was not a significant problem decades ago but nowadays applications perform a large portion of their file system interactions via memory-mapped operations rather than normal reads and writes [29]. System call replayers have non-zero overheads that do not allow them to replay high I/O rates of high-performance applications, or spikes of activity for applications that may issue most of their I/O requests at low rates.

Several researchers captured file system activity at the VFS level for Linux [2] and Windows NT [29, 37]. However, no one has replayed traces at the VFS level before.

Network tracers cannot capture the requests satisfied from the client side or file system caches. Device driver tracers capture raw disk requests and therefore cannot distinguish between file system meta-data events (e.g., pathname related calls) and data-related events. There-

fore, network level and driver level replaying are not comprehensive enough for the evaluation of an entire file system, and they are primarily suitable for replaying at the same level where they captured the trace. Nevertheless, both have their uses. For example, network trace replaying is suitable for the evaluation of NFS servers; and driver-level replayers are useful to evaluate physical disk layouts. Also, both techniques have lower overheads because they can use the CPU time that is normally spent by client-side applications to prepare events for replaying; this makes network and device-driver replayers efficient and able to replay high rates of I/O accurately.

We have designed the first VFS-level replayer which we call *Replayfs*. It replays traces captured using the Tracefs stackable file system [2]. The traces are preprocessed and optimized by our user-level trace compiler. Replayfs runs in the kernel, directly above any directory, file system, or several mounted file systems. It replays requests in the form of VFS API calls using multiple kernel threads. Replayfs's location in the kernel hierarchy allows it to combine the performance benefits of existing network and driver-level replayers with the ability to replay memory-mapped operations and evaluate entire file systems. Memory-mapped operations can be easily captured and replayed at the VFS level because they are part of the VFS API, but they are not a part of the system-call API. Replayfs uses the time normally spent on context switching, and on verifying user parameters and copying them, to prefetch and schedule future events. In addition, Replayfs saves time between requests by eliminating data copying between the kernel and user buffers.

User-mode tools cannot replay the highest possible I/O rates and spikes of such activity because due to their overheads. This is ironic because that is exactly the activity that is crucial to replay accurately. Replayfs can replay high I/O rate traces and spikes of activity even faster than the original programs that generated them, on exactly the same hardware. For example, Replayfs can replay read operations 2.5 times faster than is possible to generate them from the user level. This allows Replayfs to replay the workload accurately with the original event rates.

The rest of this paper is organized as follows. Section 2 describes our capturing and replaying design. Section 3 describes our implementation. We evaluate our system in Section 4. We describe related work in Section 5. We conclude in Section 6 and discuss future work.

2 Design

The main goal of Replayfs is to reproduce the original file system workload as accurately as possible. Memory-mapped operations can be most efficiently captured only in the kernel—they are part of the VFS API but not the system-call API. Therefore, it is logical for Replayfs to replay traces at the same level where the traces were cap-

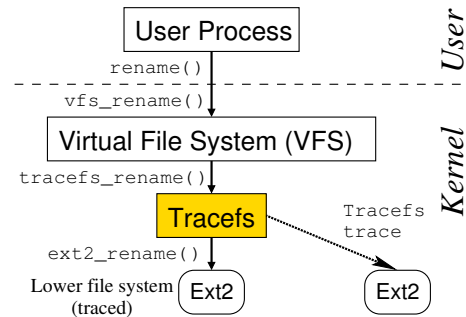


Figure 1: Tracefs is a stackable file system located above the lower file system and below the VFS.

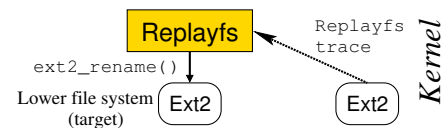


Figure 2: Replayfs is located directly above the lower file system. It is not a stackable file system. It is seen like the VFS for the lower file system.

tured. As shown in Figure 1, Tracefs [2] is a stackable file system [40]. Tracefs passes through all the *Virtual File System* (VFS) requests down to the lower file system (or a tree of several mounted file systems). Before invoking the lower operations and after returning from them, Tracefs logs the input and output values and the timing information associated with the requests. Replayfs is logically located at the same level as Tracefs—right above the lower file system as shown in Figure 2. However, Replayfs is not a stackable file system; it is not a file system either. It reproduces the behavior of the VFS during the trace capture time and appears like a VFS for the lower file system. Replayfs operates similarly to the part of the VFS which directly interacts with lower file systems.

File system related requests interact with each other and with the OS in intricate ways: concurrent threads use locks to synchronize access, they compete for shared resources such as disks, the OS may purge caches due to file system activity, etc. Therefore, to reproduce the original workload correctly it is necessary to reproduce the original timing of the requests and their side effects accurately. This is simple if the file system event rates are low. However, at high I/O rates, replayers' overheads make it more difficult to replay traces accurately. Specifically, every request-issuing process consists of three intervals: user mode activity (t_{user}), system time activity of the VFS (t_{VFS}), and the lower file system event servicing time (t_{fs}). Let us call $t_{replayer}$ the time necessary for a replayer to prepare for calling a replayed event. Clearly, if the $t_{replayer} > t_{user}$ then the timing and I/O rate could not be reproduced correctly if events are issued too close to each other as illustrated in Figure 3. This could happen, for example, if the trace is generated

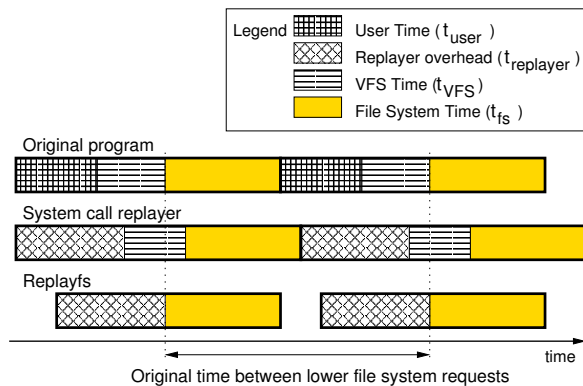


Figure 3: Two consecutive file system events triggered by an original program, a system calls replayer, and Replayfs. In this example, Replayfs and the user-mode replayer have the same per-event overheads. Nevertheless, the lower file system receives events at the same time as they were captured if replayed by Replayfs because $t_{\text{replayer}} < t_{\text{user}} + t_{\text{VFS}}$, whereas the system-call replayer is unable to generate events on time because $t_{\text{replayer}} > t_{\text{user}}$.

by a high performance application or there is a spike of I/O activity. Unfortunately, such situations are frequent and they are exacerbated because typical replayers have non-negligible overheads. Existing system-call replayers have overheads ranging from 10% [1] to 100% [7] and higher. Replayers' overheads come from the need to prefetch the data, manage threads, and invoke requests. Therefore, it is not surprising that no existing user-mode replayer can replay at the maximum possible application I/O rates and reproduce the peaks of I/O activity—the modes of file system activity that are most important for benchmarking and debugging.

Increasing the CPU or I/O speeds solves the problem for network-file-system trace replayers because the replayer and the tested file system run on different hardware. However, non-network file system trace replayers run on the same hardware. Thus, hardware changes will affect the behavior of the lower file system together with the replayer: an increase in the CPU speed can decrease both the Replayfs overheads as well as the $t_{\text{VFS}} + t_{\text{fs}}$ component. This may result in different file system request interaction, thus changing file system behavior.

Replayfs replays traces directly over the lower file system. Thus its per-operation overhead has to be smaller than $t_{\text{VFS}} + t_{\text{user}}$, not smaller than just t_{user} as illustrated by the bottom timeline of Figure 3. Therefore, if the replaying overheads of Replayfs are the same as the overheads of some user-mode replayer, then Replayfs can replay at higher I/O rates than a user-mode replayer. Running in the kernel gives Replayfs several additional advantages described in this section, resulting in lower overheads. This allows Replayfs to replay high I/O-rate traces more accurately than any system-call replayer.

2.1 Replayfs Trace

There is a natural disparity between raw traces and replayable traces. A trace captured by a tracer is often portable, descriptive, and verbose—to offer as much information as possible for analysis. A replayable trace, however, needs to be specific to the system it is replayed on, and must be as terse as possible so as to minimize replaying overheads. Therefore, it is natural to preprocess raw traces before replaying them, as shown in Figure 4. Preprocessing traces allows us to perform many tasks at the user level instead of adding complexity to the in-kernel components. We call the user mode program for conversion and optimization of the Tracefs raw traces a *trace compiler*; we call the resulting trace a *Replayfs trace*. The trace compiler uses the existing Tracefs trace-parsing library. However, new trace parsers can be added to preprocess traces that were captured using different tools, at different levels, or on different OSs. The trace compiler splits the raw Tracefs trace into three components, to optimize the run-time Replayfs operation. Each component has a different typical access pattern, size, and purpose.

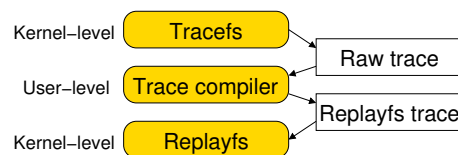


Figure 4: Captured raw traces are compiled into the Replayfs traces before replaying.

The first Replayfs trace component is called *commands*. It is a sequence of VFS operations with their associated timestamp, process ID, parameters, expected return value, and a return object pointer. At runtime, the commands are sequentially scanned and replayed one at a time. Therefore, the sequence of commands can be sequentially prefetched on demand at runtime. After the execution of every command, the actual return value is compared with the return value captured in the original trace. Replaying is terminated if the expected and actual return values do not match.

The second component is called the *Resource Allocation Table* (RAT). Because Tracefs and Replayfs operate on VFS objects whose locations in memory are not known in advance, and these objects are shared between the commands, we added a level of indirection to refer to the commands' parameters and return values. Commands contain offsets into the RAT for associated VFS objects and memory buffers. Thus, Replayfs populates RAT entries at run-time whereas the trace compiler creates commands referencing the RAT entries at trace compile time. Tracefs captures memory addresses of VFS objects related to the captured operations. All VFS ob-

jects that had the same memory address during the trace capture share the same RAT entry. The RAT is accessed randomly for reading and writing using offsets in the program elements and therefore the RAT is kept in memory during the entire replaying process. We store integer parameters together with the command stream. This allows us to decrease the size of the RAT and avoid unnecessary pointer dereferencing. Another purpose of the RAT is reference counting. In particular, the reference count of a regular VFS object may be different from the Replays reference count for the same object. For example, this happens if the object was already in use and had non-zero reference count at the time a replaying process was started. We use reference counts to release VFS objects properly upon the completion of replaying.

The third Replays trace component is the memory buffers necessary to replay the trace. They include file names and buffers to be written at some point in time. These buffers are usually accessed sequentially but some of them may be accessed several times during the replaying process. This is usually the largest component of the Replays trace. For replaying, memory buffers are accessed for reading only because the information read from the disk is discarded. We outline properties of Replays trace components in Table 1.

Component	Access	In Memory	Read/Write
Commands	Sequent.	On demand	Read only
RAT	Random	Always	Read+Write
Buffers	Random	On demand	Read only

Table 1: Replays trace components' properties.

Figure 5 shows an example Replays trace fragment. In this example, the *dentry* (Linux VFS directory entry object) RAT entry is referenced as the output object of the LOOKUP operation and as the input parameter of the CREATE operation. The "foo.bar" file name is such an example buffer.

During the Replays trace generation, the trace compiler performs several optimizations. We keep the RAT in memory during the entire replaying process. Therefore, the trace compiler reuses RAT entries whenever possible. For example, the trace compiler reuses a memory buffer entry that is not used after some point in time and stores a file pointer entry that is required only after that point. To minimize the amount of prefetching of memory buffers, the trace compiler scans and compares them. Because all the memory buffers are read-only, all except one of the buffers with exactly the same contents may be removed from the trace.

2.2 Data Prefetching

The commands and buffers components of the Replays trace are loaded into memory on demand. Because future data read patterns are known, we can apply one of

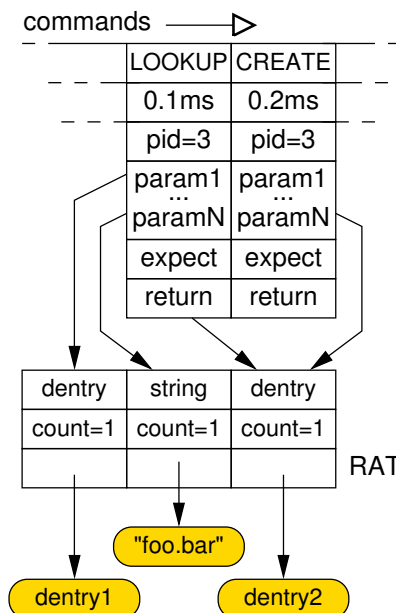


Figure 5: Example Replays trace. Commands reference the Resource Allocation Table (RAT) by the index values. The RAT points directly to the shared objects in memory.

several standard prefetching algorithms. We have chosen the *fixed horizon* algorithm [23] because it works best when the Replays trace is fetched from a dedicated disk. Therefore, we can optimize the prefetching for low CPU usage. It was theoretically shown that the fixed horizon and similar algorithms are almost optimal if the prefetching process is not I/O-bound [5]. We assume that a dedicated disk is always used for prefetching the Replays trace and therefore the prefetching process is not I/O bound. The commands and buffers Replays trace components can be located on separate disks to further decrease the I/O contention. The RAT component is always present in memory and does not interfere with the prefetching of the other two components. An additional advantage of the fixed horizon algorithm is small memory consumption. Note that the information about buffers that require prefetching is extracted from the future commands. Therefore, we prefetch the commands stream earlier than it is necessary, to keep up with the replaying of these commands.

2.3 Threads and Their Scheduling

Replays issues requests to the lower file system on behalf of different threads, if different threads generated these requests in the original trace. This is necessary to accurately reproduce and properly exercise the lower file system in case of resource contention (e.g., disk head repositioning, locks, semaphores, etc.) and to replay the timing properly if lower operations block. However, because using an excessive number of threads may hurt performance [21], Replays reuses threads if possible. In

particular, the trace compiler optimizes the commands stream: file system traces do not contain information about thread creation and termination times. Similar to the RAT entries reuse, the trace compiler reuses processes. Thus, if some program spawns a thread and after its termination it spawns another one, Replayfs will automatically use one thread to replay operations invoked by both of them. To minimize the scheduling-related overheads, Replayfs does not create a master thread to manage threads. For example, there is only one thread running if the traced data was generated by a single process. It is important to note that the scheduling overheads during replaying are approximately the same as during the trace capture time. This is one of the conditions that is necessary to replay traces efficiently on the same hardware as was used during the trace capture.

Standard event timers have a precision of about 1ms. To increase the event replaying precision, a *pre-spin* technique is commonly used [1,9]: event timers are set to about 1ms before the actual event time. The awoken thread then spins in a loop, constantly checking the current time until the desired event time is reached. A natural way to lower the CPU load is to use the pre-spinning time for some other productive activity. We call this technique a *productive pre-spin*. Replayfs uses it to move another thread into the run-queue if there is enough time before the actual event time and the next operation has to be replayed by a different thread. The next thread is not woken up immediately; it is just put on the run-queue. This way CPU cycles are more effectively spent on moving the process into a run-queue instead of spinning.

2.4 Zero Copying of Data

One of the main advantages of kernel replayers over user mode replayers is the ability to avoid copying of unnecessary data across the kernel-user boundary. Thus, data from pages just read does not need to be copied to a separate user mode buffer. The data read during the trace replaying is of no interest to the replaying tools. If desired, checksums are sufficient for data verification purposes. Instead of copying we read one byte of data from a data page to set the page's accessed bit. However, there is no easy way a user-mode program can read data but avoid copying it to user space. Direct I/O allows programs to avoid extra data copying but is usually processed differently at the file system level and therefore the replaying would be inaccurate if normal read or write requests are replayed as direct I/O requests.

Avoiding the data copying is more difficult for write operations. However, kernel-mode replayers have access to low-level file system primitives. For example, a data page that belongs to the trace file can be simply moved to the target file by just changing several pointers. Therefore, even for writing, most data copying can be elimi-

nated. Elimination of unnecessary data copying reduces the CPU and memory usage in Replayfs. Note that user-mode replayers that do not use direct I/O for fetching the data from the disk, have to copy the data *twice*: first, they copy it from the kernel to the user space buffers when they load the trace data; then they copy the data to the kernel when they issue a write request.

2.5 File System Caches

File system page caches may be in a different state when replaying the traces than when capturing them. Some times it is desirable to replay a trace without reproducing the original cache state precisely; this is useful, for example, when replaying a trace under different hardware conditions (e.g., for benchmarking). However, sometimes (e.g., for debugging or forensics) it is desirable to reproduce the lower file system behavior as close to the original as possible. Therefore, Replayfs supports three replaying modes for dealing with read operations. First, reads are performed according to the current cache state. In particular, Replayfs calls all the captured buffer read operations. In this case, only non-cached data pages result in calls to page-based read operations. Second, reads are performed according to the original cache state. Here, reads are invoked on the page level only for the pages that were not found in the cache during tracing. Third, reads are not replayed at all. This is useful for recreation of the resulting disk state as fast as possible.

Directory entries may be released from the dentry cache during the replaying process but stay in during the trace capture. This can result in an inconsistency between the RAT entries and the actual dentries. To avoid this situation we force the dentries that stayed in the cache during the capture to stay in the cache during the replaying process: we increase a dentry's reference counter every time it is looked up and decrease it when dentries were released according to the original trace.

2.6 Asynchronous File System Activity

Some of the file system activity is performed asynchronously by a background thread. Replying asynchronous activity is complicated because it is intertwined with file system internals. For example, access-time updates may be supported on the file system used for replaying but not be supported on the original one. Therefore, Replayfs replays such activity indirectly: the meta-data information is updated on time according to the trace data but it is up to the lower file system how and when to write the corresponding changes to stable storage. This way the replaying process exercises the lower file system without enforcing restrictions that are related only to the originally traced file system.

2.7 Initial File System State

In the simplest case, a trace can be captured starting with an empty file system and then replayed over an empty file system. However, traces usually contain operations on files and other file system objects which existed before the tracing process was started. Therefore, a file system must be prepared before the replaying may begin. It is convenient to prepare the file system using the information contained in the trace. However, the best way to prepare the lower file system is snapshotting [13, 26, 39]. Full restoration of the initial file system state makes trace replaying more precise because many file system algorithms have different performance with different file system states. For example, directory sizes may influence the performance of `LOOKUP` and `REaddir` operations even if most of the files in the directory never show up in a trace. Existing snapshotting systems can capture and restore snapshots for `Replayfs`.

3 Implementation

Before a file system trace can be precisely replayed, it has to be captured without perturbing the behavior of the lower file system. Therefore, we performed several optimizations in `Tracefs`. Traditionally, stackable file systems buffer data twice. This allows them to keep both modified (e.g., encrypted or compressed) and unmodified data in memory at the same time and thus save considerable amounts of CPU time and I/O. However, `Tracefs` does not modify the data pages. Therefore, double caching does not provide any benefits but makes the page cache size effectively half its original size. The data is copied from one layer to the other, unnecessarily consuming CPU resources. Unfortunately, the Linux VFS architecture imposes constraints that make sharing data pages between lower and upper layers complicated. In particular, a data page is a VFS object that belongs to a single inode and uses the information of that inode at the same time [12].

We applied a solution used in `RAIF` [15]. Specifically, data pages that belong to the upper inode are assigned to lower-level inodes for the short duration of the lower-level page-based operations. We tested the resulting `Tracefs` stackable file system on a single-CPU and on multi-CPU machines under compile and I/O-intensive workloads. In addition to the CPU time and memory savings, this optimization allowed us to reduce the `Tracefs` source size by about 250 lines.

In most cases, `Replayfs` does not need the original data buffers for replaying `READ` operations. Even for data verification purposes, an MD5 checksum is sufficient. Therefore, we added a new `Tracefs` option that instructs it to capture the data buffers for writing operations but not for reads. This allowed us to reduce both the `Tracefs` trace sizes and the `Tracefs` system time overheads.

Trace compiler. The trace compiler is optimized for performance. Its intermediate data sets are commonly larger than the amount of the available memory. Therefore, we added several hash data structures to avoid repeatedly scanning the data and thus reduce I/O usage. We compare the buffers by comparing their MD5 checksums. This allows us to save the CPU time because MD5 checksums are calculated only once for every buffer. The trace compiler consists of 4,867 lines of C code.

Replayfs kernel module. Because the trace compiler prepares the data for replaying, `Replayfs` itself is relatively small and simple. It consists of thread management, timing control, trace prefetching and eviction, operation invocation, and VFS resource-management components. `Replayfs`'s C source is 3,321 lines long. `Replayfs` supports accelerated or decelerated playback by a fixed factor, as well as replaying as fast as possible.

Both `Replayfs` and `Tracefs` are implemented as loadable kernel modules. We have ported `Tracefs` to the 2.6 Linux kernel and now both `Tracefs` and `Replayfs` can be used on either 2.4 or 2.6 Linux kernels.

4 Evaluation

We conducted our benchmarks on a 1.7GHz Pentium 4 machine with 1GB of RAM. Its system disk was a 30GB 7200 RPM Western Digital Caviar IDE formatted with Ext3. In addition, the machine had two Maxtor Atlas 15,000 RPM 18.4GB Ultra320 SCSI disks formatted with Ext2. We used one of the SCSI disks for storing the traces and the `Replayfs` traces; we used the other disk for running the test workloads and replaying them. We remounted the lower file systems before every benchmark run to purge file system caches. We ran each test at least ten times and used the Student-*t* distribution to compute the 95% confidence intervals for the mean elapsed, system, user, and wait times. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. In each case, the half-widths of the confidence intervals were less than 5% of the mean. The test machine was running a Fedora Core 3 Linux distribution with a vanilla 2.6.11 kernel.

4.1 Evaluation Tools and Workloads

We created one additional statistics module, for evaluation purposes only: this module records the timeline statistics from Ext2 and the timing-deviation figures from `Replayfs`. This module uses the `/proc` interface to export the data to the user-level for analysis and plotting. The statistics module stores the resulting information in a static array and the only effects to a file-system operation are querying the time and incrementing a value in the output array. Therefore, the corresponding overheads were negligible: we measured them to be below 1% of the CPU time for all the experiments we ran.

Am-utils build. Building Am-utils is a CPU-intensive benchmark. We used Am-utils 6.1 [25]: it contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Though the Am-utils compile is CPU intensive, it contains a fair mix of file system operations. According to the instrumented Ext2 file system, it uses 25% writes, 22% lseek operations, 20.5% reads, 10% open operations, 10% close operations, and the remaining operations are a mix of REaddir, LOOKUP, etc. We used Am-utils because its activity is not uniform: bursts of I/O activity are separated by intervals of high CPU activity related to the user mode computations. This allows us to analyze the replaying precision visually. Also, the compilation process heavily uses the memory-mapped operations.

Postmark. Postmark v1.5 [16] simulates the operation of electronic mail servers. It performs a series of file system operations such as appends, file reads, creations, and deletions. This benchmark uses little CPU but is I/O intensive. We configured Postmark to create 20,000 files, between 512–10K bytes in size, and perform 200,000 transactions. We selected the create, delete, read, and write operations with equal probability. We used Postmark with this particular configuration because it stresses Replays under a heavy load of I/O-intensive operations.

Pread. Pread is a small micro-benchmark we use to evaluate Replays’s CPU time consumption. It spawns two threads that concurrently read 1KB buffers of cached data using the `pread` system call. In every experiment, Pread performed 100 million read operations. We use Pread to compare our results with Buttress [1], a state-of-the-art system call replayer that also used `pread`. This micro-benchmark also allowed us to demonstrate the benefits of our zero-copying replaying.

4.2 Memory Overheads

The memory consumed by replayers effectively reduces the file system cache sizes and therefore can affect the behavior of the lower file system. The compiled binary module sizes are negligible. They are 29KB for the Replays module and 3KB for the statistics module. Our user mode trace compiler reduces the trace size by generating the variable length program elements and by eliminating duplicate data buffers. Table 2 shows some characteristics of the raw and compiled traces as well as their compilation times. We can see that the original Am-utils trace size was reduced by 56%, by 70% for Postmark, and by 45% for Pread. Recall that only the RAT is entirely kept in memory and its size was small for all the traces we used. The program and the buffers trace com-

ponents are prefetched on demand. We used a separate disk for storing the traces. This reduced I/O contention and allowed us to prefetch the minimal amount of data that is necessary to replay the trace on time. In addition, direct access to the page cache allowed us to promptly evict pages that will not be accessed in the near future. As a result, the memory used for prefetching and storing the traces never exceeded 16MB for all of our experiments. This means that all the Replays memory overheads together were less than 2% of the available memory on our test machine during any time of our benchmark runs.

	Am-utils	Postmark	Pread
Raw trace	334 MB	3,514 MB	7,248 MB
Commands	25 MB	224 MB	4,000 MB
RAT	0.4 MB	3.3 MB	60 bytes
Buffers	122 MB	832 MB	3 bytes
Compilation time (minutes)	<1	15	31

Table 2: Size and compilation time of the traces.

4.3 Timing Precision of Replaying

Standard OS timers usually have low resolution. For example, standard Linux timers have a resolution of one millisecond which is much larger than a microsecond, the typical duration of a file system operation that does not result in I/O. We have applied the pre-spin technique [9] described in Section 2 to bring the timing accuracy to the microsecond scale. Figure 6 shows the cumulative distribution function (CDF) of the operation invocation timing errors. Naturally, the timing errors of a Postmark run with no pre-spin are distributed almost equally between 0 and 1 millisecond because events are triggered with the poor millisecond resolution. We can see that pre-spinning dra-

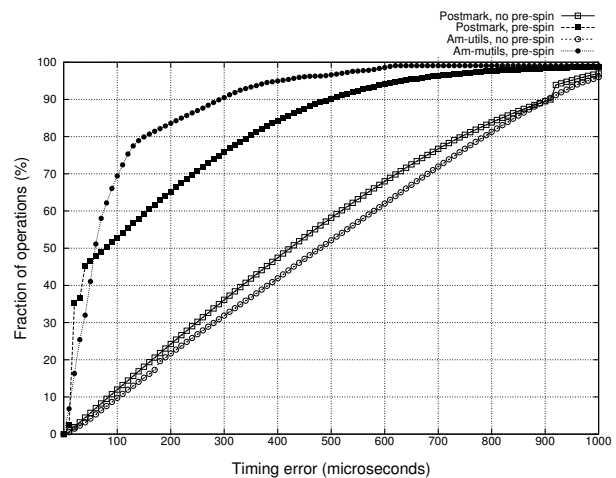


Figure 6: Cumulative distribution functions of the event invocation error rates during several replaying experiments. The closer the curve is to the upper-left corner, the better the average accuracy is.

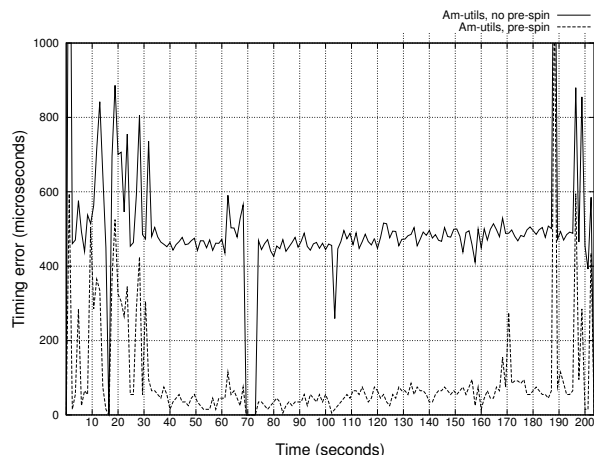


Figure 7: Dependence of the average invocation time error on elapsed time during the Am-utils trace replaying.

matically decreases the error values. However, the error distributions differ for different workloads.

Two figures clarify this behavior. Figure 7 shows the average timing error during every second of the Am-utils trace replaying. Figure 8 shows the corresponding file system operation counts recorded by the instrumented Ext2. We can see a clear correlation between the replaying event rates and the related average error. The reason behind this correlation is that events well spaced apart are replayed with high accuracy, whereas events that should be invoked close to each other cannot be invoked as accurately because of the I/O and CPU time overheads. Therefore, we can make two conclusions.

First, the CDF of invocation errors can easily hide real replaying problems. For example, a CDF captured with a slow operation rate workload may indicate that almost all of the operations were replayed with high precision. However, Figure 6 shows that no information can be inferred about how the same tool would behave at medium or high I/O rates.

Second, the timers' accuracy is not as important for file system activity replayers as it was believed before [1, 9]. The timer's resolution contributes to the event invocation errors only at the low event rates where timing precision is not even necessary. On one hand, it is unlikely that events separated by long intervals of no file system activity (as long as several durations of an average event) will influence each other too much. On the other hand, file system operations invoked close to each other, and especially if they are invoked by different processes and they overlap, are more likely to interfere with each other. Therefore, it is desirable to replay them as precisely as possible. However, in that case the timer's resolution has small overall impact on the resulting timing quality. Instead, the overheads that replayers add between operations define the timing precision as we discussed in Sec-

tion 2. We can see that in Figure 8 by comparing the traces of an Am-utils replayed with different timer resolutions. We cannot easily see the pre-spinning improvement effects in Figure 8, because they are only visible at the micro-second resolution; Figure 6 shows our pre-spinning improvements more prominently. However, we can see from Figure 8 that in both timer resolution cases there are small discrepancies at the peaks of activity between the replayed and captured traces.

One may assume that an increase in the CPU speed can solve the timing precision problem. This is indeed the case for network packet and network file system replayers because the replayer and the target system run on different machines. However, this is not the case for non-network file system replayers because they execute on the same machine as the tested file system. Therefore, with a faster CPU, the replayed operations will also execute faster and their corresponding interactions will change disproportionately; that is the portion of the CPU time spent servicing file system requests will decrease and requests from different processes will overlap less, thus processes will compete less for locks or disk heads.

4.4 CPU Time Consumption

System-call replaying tools run in user mode and invoke the same system calls that were invoked by the original user programs. Usually user-level replayers have CPU time overheads that are higher than the user activity intervals in the original trace. Replayfs runs in the kernel and therefore avoids wasting time on some of the operations required to cross the user-kernel boundary.

Let us consider the `pread` system call. After it is invoked, the VFS converts the file descriptor number to an in-kernel file structure, checks parameters for validity and correspondence with the file type, and verifies that the buffer can be used for writing. Replayfs benefits from the following four optimizations: (1) kernel mode VFS objects are readily available to Replayfs and need not be looked up; (2) Replayfs operates on VFS objects directly and the file structure argument is taken directly by looking up the RAT; (3) parameters and file access modes were checked during the trace capture and can be skipped; and (4) memory buffers are not passed from the user space, so Replayfs can allocate them directly without having to verify them. Figure 9 shows the times related to the execution of the original `Pread` program and replaying its trace by Replayfs at full speed. The *Replayfs* bar in Figure 9 shows that skipping the VFS operations described above allows Replayfs to replay the `Pread` trace 32% faster than the original program generated it on the same hardware.

Replayfs can also avoid copying data between user-mode buffers and the kernel pages. The *Replayfs-nocopy* bar in Figure 9 demonstrates that

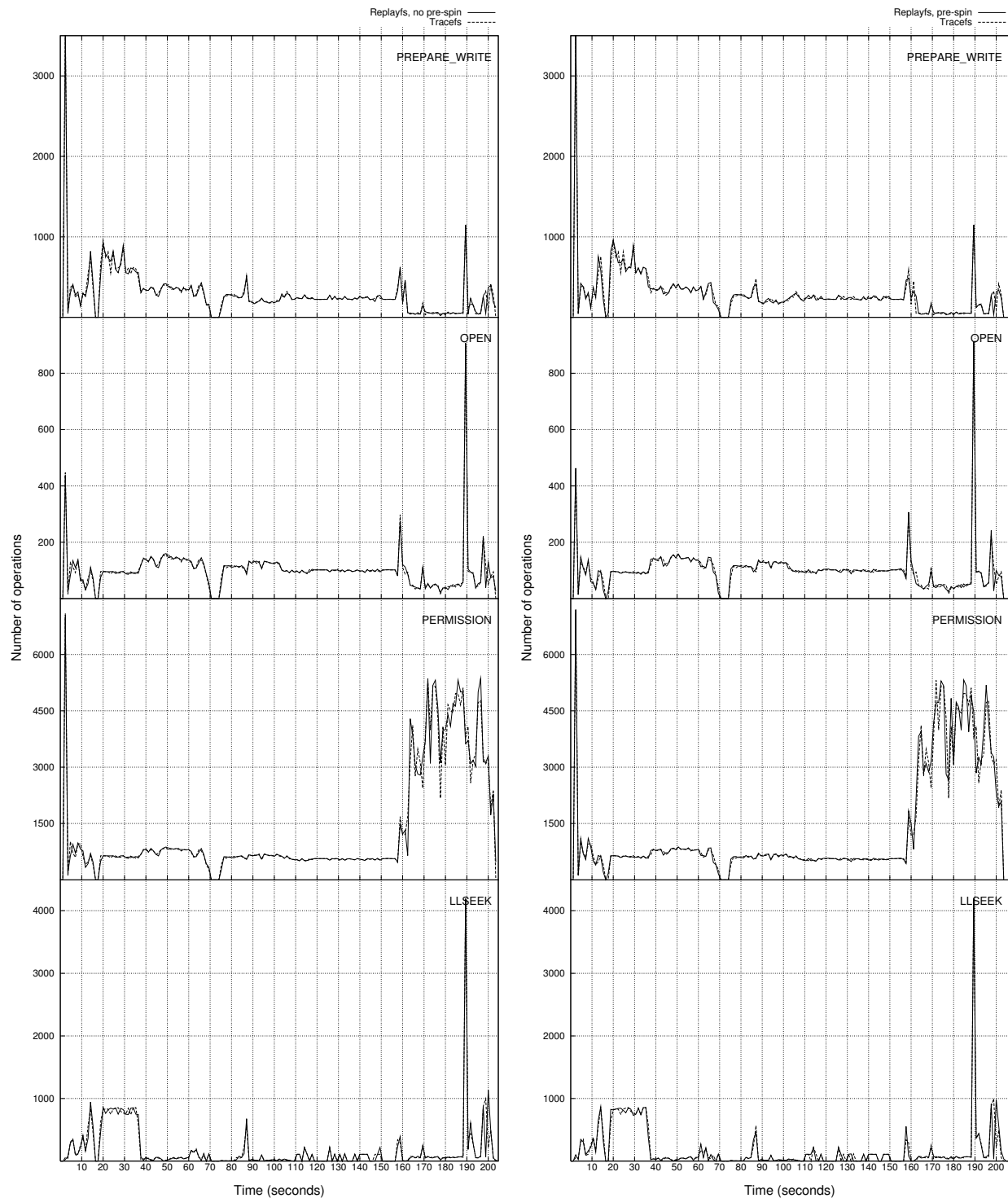


Figure 8: Counts of the file system operations as seen by the lower Ext2 file system while replaying the Am-utils traces without the pre-spin timer enhancement (left) and with the pre-spin enhancement (right). As we can see, there is no clear difference between the two on the seconds scale in spite of the fact that timing on the micro-second scale is better with the pre-spin configuration. In most cases the Replays and Tracefs curves overlap and are indistinguishable. Small timing discrepancies are correlated with peaks of I/O activity. We show four operations with the highest peaks of activity (>800 operations) because they have the highest timing errors observed. Also, we do not show the RELEASE and COMMIT_WRITE operations because their graphs closely resemble the shapes of the OPEN and PRPARE_WRITE operations, respectively.

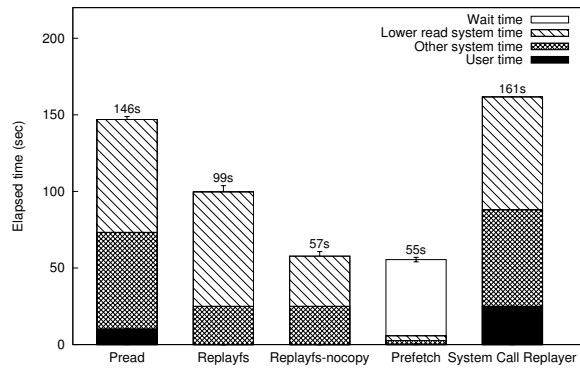


Figure 9: Elapsed times of our *Pread* program (*Pread*), the *Pread* trace replayed by *Replayfs* while copying the data just read (*Replayfs*), elapsed time of the *Pread* trace replayed by a *Replayfs* that skips the data copying (*Replayfs-nocopy*), the time necessary to read the trace data as fast as possible (*Prefetch*), and estimated elapsed time of the system call replayer with a 10% user time overhead (*System Call Replayer*)

Replayfs can replay the original *Pread* trace 61% (2.5 times) faster than the original program generated it on the same hardware. We can see that the data copying alone can reduce the execution of the file system read operation by 55% (685 out of 1,240 CPU cycles on average).

In the case of the *Pread* trace, no prefetching of data was necessary (except for 3 bytes of null-terminated file name). We created a modified version of the *Pread* program we call *Prefetch* that sequentially reads data from a file as fast as possible. It took *Prefetch* 55 seconds on average to read the *Pread* program trace component. The *Prefetch* bar in Figure 9 shows that out of these 55 seconds, 49 were spent waiting for I/O completion. This means that the replaying process was not I/O-bound because *Replayfs* prefetches traces asynchronously. However, a further decrease of the *Replayfs* CPU overheads may make *Replayfs* I/O-bound while replaying the *Pread* trace: *Replayfs* would reach the physical limitations of the I/O hardware and the disks, at which point replaying could not be sped up further. In that case, this problem can be resolved by replacing the disk drive or its controller because the tested file system and the *Replayfs* traces are located on different physical drives.

We compared *Replayfs* with the state-of-the-art *Buttress* system call replayer. Unfortunately, *Buttress* availability for public, especially for comparison purposes, is limited and we could not evaluate it. Its overheads under the *Pread* workload are reported to be 10% [1]. The rightmost bar in Figure 9 represents an extrapolated timing result for *Buttress*; for visual comparison purposes, we created that bar by adding 10% overhead of the elapsed time to the user time of the original *Pread* program time. Note that the actual overhead value is not as important as the fact that the overhead is positive. Be-

cause the overhead is positive, user-level replayers cannot replay traces like *Pread* at the same rate as the original *Pread* program can issue I/O requests. Having low or even negative overheads in *Replayfs* results in good reproduction of the original timing conditions. Despite some existing discrepancies between the original traces and the replayed ones (seen in Figure 8), the replayed and the original figures overlap for most of the operations even during peaks of activity. Existing system call replaying tools such as *Buttress* cannot match the trace as closely because of their inherent overheads. *Buttress* is a well designed system; its overheads are lower than the overheads of other published systems that replay traces via system calls. However, it is precisely because *Buttress* and similar systems run in the user level that they have higher overheads, which in turn imposes greater limitations on their ability to replay traces accurately.

Another benefit of *Replayfs*'s low overheads is the ability to replay the original traces at faster speeds even on the same hardware. As we described above, we can replay read-intensive traces 2.5 times faster than their original time. In addition, we replayed the *Am-utils* trace in the accelerated mode. We were able to replay the 210-second long *Am-utils* trace in under 6 seconds, reproducing all the memory-mapped and other disk state changing operations. This represents a speedup of more than two orders of magnitude.

5 Related Work

Trace capture and replaying have been used for decades and we describe only a representative set of papers related to file system activity tracing and replaying.

Capturing traces. We describe tracers according to the level of abstraction where they capture the file system activity: system-call-level tracers, virtual file system level tracers, network tracers, and finally driver-level tracers. We discuss them in this order because network-level tracers capture file system information at a level of abstraction that is higher than the driver-level, but is lower than the VFS-level.

- The most common tool used to capture system calls is *strace* [38]. It uses the *ptrace* system call to capture the sequence of system calls invoked by an application together with associated parameter values. *DFSTrace* showed that special measures have to be taken to collect file system traces in distributed environments during long intervals of time, to minimize the volume of generated and transferred data [19]. The problem of missed memory-mapped operations in system call traces has long been recognized [22]. However, only in 2000 did Roselli show that unlike decades ago, memory-mapped I/O operations are now more common than normal reads and writes [29].

- Others collected traces at the virtual file system level for Linux [2] and Windows NT [29, 37]; these traces include memory-mapped operations.
- Network packet traces can be collected using specialized devices or software tools like *tcpdump* [11]. Specialized tools can capture and preprocess only the network file system related packets [4, 8]. Network file system traces do not contain information about the requests satisfied from the caches but can contain information about multiple hosts.
- Driver-level traces contain only the requests that are not satisfied from the caches. This is useful when disk layout information needs to be collected while minimizing the trace size [30].

Trace replaying. Similar to capturing traces, replaying traces is performed at several logical levels. Usually, the traces are replayed at the same level that they were captured from. This way changes to the timing and operations mix are minimized. However, for simplicity, some authors replay kernel-level traces at the user level.

- It is simple to replay system calls that contain all the necessary information as parameters. Several existing system call replayers are designed specifically to replay file system activity. *Buttress* [1] and *DFSTrace* [19] can replay system call traces from the user level. Buttress's evaluation showed a 10% slowdown if replaying traces at high I/O rates, which the authors claimed was "accurate enough." Performance data for DFSTrace's replaying mode is not available, mostly because the main focus of the authors was on capturing traces.
- Network traffic replayers operate in user mode and can replay arbitrary network traces [9, 33]. Network file system trace replaying is conceptually similar to ordinary network packet trace replaying. However, knowledge of the network file system protocol details allows replayers to reorder some packets for faster replaying [42]. Replayers and tracers can run on dedicated machines separate from the tested servers. Thus, network file system trace replaying is the least intrusive replaying and capturing method.
- Replying I/O patterns at the disk-driver level allows the evaluation of elevator algorithms and driver subsystems with lower overheads and little complexity. Also, it allows the evaluation of disk layouts. For example, Prabhakaran et al. used a driver level replayer to measure the effects of the journal file relocation on the disk [27]. In this particular case, system-call-level replaying was not appropriate because the physical file's location on the disk could not be easily controlled from the user level.
- Others capture and then replay traces at different logical levels. For example, *Drive-Thru* [24] pro-

cesses driver-level traces and replays them at the system-call level to evaluate power consumption. Unrelated file system operations are removed during the preprocessing phase to speed up the replaying process. Several others replayed network file system traces in disk simulators for benchmarking [36, 41]. Network traces are most suitable for this purpose because they are captured below caches and thus minimally disturb the workload.

File system state versioning. File system trace replaying can be considered a form of fine-grained versioning [20, 32]. Replying can reproduce the version of the file system state including possible state abnormalities caused by timing conditions. This property is useful for forensics (post-attack investigation) and debugging purposes. Also, it can be used to emulate the aging of a file system before running actual benchmarks [31].

Before replaying file system activity, replayers may have to recreate the pre-tracing file system image. This is important for accuracy: file layouts and the age of the file system can affect its behavior significantly. Some authors have opted to extrapolate the original file system state based on information gleaned from the trace alone [19, 42]. This technique has three disadvantages. First, full path name information is required in the trace data to identify the exact directories in which files were accessed [22]. Second, files that were not accessed during the trace are not known to the system, and those files could have affected the file system's layout and age. Third, several trace-capture techniques omit information that is vital to replaying accurately. For example, an NFS (v2 and v3) trace replayer that sees an NFS_WRITE protocol message cannot tell if the file being written to existed before or not. It is therefore our belief that the best method to restore the pre-tracing file system state is to use snapshotting [13, 26, 39].

Data prefetching. Data prefetching is a common technique to decrease application latency and increase performance [3]. Future access patterns are normally inferred from a history of past accesses or from hints provided by applications [34]. If access patterns are known in advance, two simple approaches are possible. First, data can be aggressively read in advance without overwriting the already prefetched data. Second, data can be read just in time to avoid stalls. Cao et al. showed that both approaches are at most two times worse than the optimal solution [5]. Both algorithms have simple implementations. The TIP2 system [23] uses a version of the second algorithm called *fixed horizon*. A more sophisticated *reverse aggressive* [17] algorithm has near-optimal performance but is difficult to implement. The *forestall* [18] algorithm is an attempt to combine the best of these algorithms: simplicity and prefetching performance.

Timing inaccuracy. Existing system-call replayers suffer from timing precision problems and peak-load reproduction problems to some degree, for several reasons:

- User mode replayers have high memory and CPU overheads due to redundant data copying between user and kernel buffers [28].
- Page eviction is not completely controlled from the user level and thus prefetching policies are harder to enforce. Nevertheless, the `madvise` interface can help somewhat to control data page eviction [9].
- Some kernels are not preemptive and have long execution paths including in interrupt handlers [10].
- Replaying processes can be preempted by other tasks. This can be partially solved by instructing the scheduler to treat the replaying process as real time process [9].
- Standard timer interfaces exposed to the user level are not precise enough. Several authors investigated this problem and came to similar conclusion [1, 9]: it is sufficient to setup the timer early and busy-wait only after the timer expires.

The metric used to evaluate the replaying precision in several papers is the average difference between the actual event time and the traced event time [1, 9]. For example, using better kernel timers and the `madvise` interface resulted in a typical 100-microsecond of difference [9]—almost a 100 times improvement compared with a replayer without these measures [33].

6 Conclusions

Trace replaying offers a number of advantages for file system benchmarking, debugging, and forensics. To be effective and accurate, file system traces should be captured and replayed as close as possible to the file system code. Existing systems that capture file system traces at the network file system level often miss on client-side cached or aggregated events that do not translate into protocol messages; system-call traces miss the ever more popular memory-mapped reads and writes; and device-driver level traces omit important meta-data file system events such as those that involve file names. These problems are exacerbated when traces that were captured at one level are replayed at another: even more information loss results.

We demonstrated that unlike previously believed, the accuracy of replaying high I/O-rate traces is limited by the overheads of the replayers—not the precision of the timers. Since most file systems run in the kernel, user-level file system replayers suffer from overheads that affect their accuracy significantly. User-mode replayers produce an excessive number of context switches and data copies across the user-kernel boundary. Therefore, existing replayers are inaccurate and unable to replay file system traces at high event rates.

We have designed, developed, and evaluated a new replaying system called `Replayfs`, which replays file system traces immediately above file systems, inside the kernel. We carefully chose which actions are best done offline by our user-level trace compiler, or online by our runtime kernel `Replayfs` module. Replaying so close to the actual file system has three distinct benefits:

- First, we capture and replay all file system operations—including important memory-mapped operations—resulting in more accurate replaying.
- Second, we have access to important internal kernel caches, which allowed us to avoid unnecessary data copying, reduce the number of context switches, and optimize trace data prefetching.
- Third, we have precise control over thread scheduling, allowing us to use the oft-wasted pre-spin periods more productively—a technique we call *productive pre-spin*.

Our kernel-mode replayer is assisted by a user-mode trace compiler, which takes portable traces generated by `Tracefs`, and produces a binary replayable trace suitable for executing in the kernel. Our trace compiler carefully partitions the data into three distinct groups with different access patterns, which allowed us to apply several optimizations aimed at improving performance:

Commands which are read sequentially;

Resource Allocation Table (RAT) which determines how in-memory resources are used throughout the replaying phase. In particular, the RAT allows us to reuse resources at replay time once they are no longer needed, rather than discarding them;

Buffers which are bulk I/O data pages and file names that are often accessed randomly on a need basis.

This partitioning and the possibility to evict cached data pages directly allowed us to reduce memory usage considerably: in all of our experiments, `Replayfs` consumed no more than 16MB of kernel memory, which is less than 2% on most modern systems. Overall, `Replayfs` can replay traces faster than any known user-level system, and can handle replaying of traces with spikes of I/O activity or high rates of events. In fact, thanks to our optimizations, `Replayfs` can replay traces captured on the same hardware—faster than the original program that produced the trace.

6.1 Future Work

Commands executed by different threads may be issued out of their original order. For example, if one thread is waiting for a long I/O request, other threads may continue their execution unless there is a dependency between requests. This is useful for stress-testing and certain benchmarking modes. However, commands have to be synchronized at points where threads depend on each

other. For example, if an original trace shows that one thread read a file after a second thread wrote the same file, then this ordering should be preserved during trace replaying. We are modifying our existing trace compiler to add thread synchronization commands to the commands Replayfs trace component.

Some workloads may result in different behavior of the file system even if the operations' order is preserved. For example, two threads concurrently writing to the same file may create a different output file due to in-kernel pre-emption. Future policies will allow Replayfs to differentiate between real replaying errors and tolerable mismatches between return values due to race conditions.

There is a large body of existing traces which were captured over the past decades on different systems or at different levels. Unfortunately, many of these traces cannot be replayed for lack of tools. We are currently developing user-mode translators which can convert such traces from other formats into our own portable format.

We have carefully analyzed the VFS interfaces of Linux, FreeBSD, Solaris, and Windows XP. Despite their significant internal implementation differences, we found them to be remarkably similar in functionality. This is primarily because file system interfaces have evolved over time to cooperate best with APIs such as the POSIX system-call standard. Therefore, we also plan to port Replayfs to other operating systems.

7 Acknowledgments

We would like to acknowledge Akshat Aranya, Jordan Hoch, and Charles P. Wright for their help at different stages of Replayfs's design, development, and testing. We would also like to thank all FSL members for their support and a productive environment. This work was partially made possible by NSF awards EIA-0133589 (CAREER) and CCR-0310493, and HP/Intel gifts numbers 87128 and 88415.1.

References

- [1] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A Toolkit for Flexible and High Fidelity I/O Benchmarking. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 45–58, San Francisco, CA, March/April 2004.
- [2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.
- [3] L. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal*, 5(2):78–101, 1966.
- [4] M. Blaze. NFS Tracing by Passive Network Monitoring. In *Proceedings of the USENIX Winter Conference*, San Francisco, CA, January 1992.
- [5] P. Cao, E. Felten, A. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 188–197, Ottawa, Canada, May 1995.
- [6] P. M. Chen and D. A. Patterson. A New Approach to I/O Performance Evaluation - Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 1–12, Seattle, WA, May 1993.
- [7] F. Cornelis, M. Ronsse, and K. Bosschere. Tornado: A novel input replay tool. In *Proceedings of the 2003 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '03)*, volume 4, pages 1598–1604, Las Vegas, Nevada, June 2003.
- [8] D. Ellard and M. Seltzer. New NFS Tracing Tools and Techniques for System Analysis. In *Proceedings of the Annual USENIX Conference on Large Installation Systems Administration*, San Diego, CA, October 2003.
- [9] W. Feng, A. Goel, A. Bezzaz, W. Feng, and J. Walpole. Tcpiro: a high-performance packet replay engine. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 57–64, Karlsruhe, Germany, 2003.
- [10] A. Goel, L. Abeni, J. Snow, C. Krasic, and J. Walpole. Supporting Time-Sensitive Applications on General-Purpose Operating Systems. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI '02)*, pages 165–180, Boston, MA, December 2002.
- [11] LBNL Network Research Group. The TCP-Dump/Libpcap site. www.tcpdump.org, February 2003.
- [12] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proceedings of Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 3–6, Copper Mountain Resort, CO, December 1995.
- [13] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994.
- [14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
- [15] N. Joukov, A. Rai, and E. Zadok. Increasing distributed storage survivability with a stackable raid-like file system. In *Proceedings of the 2005 IEEE/ACM Workshop on Cluster Security, in conjunction with the Fifth IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, Cardiff, UK, May 2005 (**Won best paper award**).

- [16] J. Katcher. PostMark: A New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. www.netapp.com/tech_library/3022.html.
- [17] T. Kimbrel and A. Karlin. Near-optimal Parallel Prefetching and Caching. In *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, pages 540–549, October 1996.
- [18] T. Kimbrel, A. Tomkins, R. Patterson, B. Bershad, P. Cao, E. Felten, G. Gibson, A. Karlin, and K. Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI 1996)*, pages 19–34, Seattle, WA, October 1996.
- [19] L. Mummert and M. Satyanarayanan. Long term distributed file reference tracing: Implementation and experience. Technical Report CMU-CS-94-213, Carnegie Mellon University, Pittsburgh, PA, 1994.
- [20] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004.
- [21] J. Ousterhout. Why Threads are a Bad Idea (for most purposes). In *Invited Talk at the 1996 USENIX Technical Conference*, January 1996. home.pacbell.net/ouster/threads.ppt.
- [22] J. Ousterhout, H. Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 15–24, Orcas Island, WA, December 1985.
- [23] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, CO, December 1995.
- [24] D. Peek and J. Flinn. Drive-Thru: Fast, Accurate Evaluation of Storage Power Management. In *Proceedings of the Annual USENIX Technical Conference*, pages 251–264, Anaheim, CA, April 2005.
- [25] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. www.am-utils.org.
- [26] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005.
- [27] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the Annual USENIX Technical Conference*, Anaheim, CA, May 2005.
- [28] A. Purohit, J. Spadavecchia, C. Wright, and E. Zadok. Improving Application Performance Through System Call Composition. Technical Report FSL-02-01, Computer Science Department, Stony Brook University, June 2003. www.fsl.cs.sunysb.edu/docs/cosy-perf/.
- [29] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proc. of the Annual USENIX Technical Conference*, pages 41–54, San Diego, CA, June 2000.
- [30] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. In *Proceedings of the Winter USENIX Technical Conference*, pages 405–420, San Diego, CA, January 1993.
- [31] K. A. Smith and M. I. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 203–213, Seattle, WA, June 1997.
- [32] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata Efficiency in Versioning File Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, San Francisco, CA, March 2003.
- [33] tcpreplay developers. *tcpreplay(8)*, February 2004. tcpreplay.sourceforge.net.
- [34] A. Tomkins, R. Patterson, and G. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114, Seattle, WA, June 1997.
- [35] Transaction Processing Performance Council. Transaction Processing Performance Council. www.tpc.org, 2005.
- [36] M. Uysal, A. Merchant, and G. A. Alvarez. Using MEMS-Based Storage in Disk Arrays. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 89–101, San Francisco, CA, March 2003.
- [37] W. Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 93–109, Charleston, SC, December 1999.
- [38] W. Akkerman. strace software home page. www.liacs.nl/~wichert/strace/, 2002.
- [39] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Computer Science Department, Stony Brook University, October 2004. www.fsl.cs.sunysb.edu/docs/unionfs-tr/unionfs.pdf.
- [40] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.
- [41] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 289–304, Monterey, CA, January 2002.
- [42] N. Zhu, J. Chen, T. Chiueh, and D. Ellard. Scalable and Accurate Trace Replay for File Server Evaluation. Technical Report TR-153, Computer Science Department, Stony Brook University, December 2004. www.ecsl.cs.sunysb.edu/tr/TR153.pdf.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *login:*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications; see <http://www.usenix.org/membership/specialdisc.html>

SAGE, The System Administrators Guild

SAGE is a Special Interest Group (SIG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

For more information about membership, conferences, or publications, see <http://www.usenix.org>.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

Addison-Wesley Professional/Prentice Hall Professional • AMD • Asian Development Bank
Cambridge Computer Services, Inc. • EAGLE Software, Inc. • Electronic Frontier Foundation
Eli Research • GroundWork Open Source Solutions • Hewlett-Packard • IBM • Intel • Interhack
The Measurement Factory • Microsoft Research • NetApp • Oracle • OSDL • Perfect Order
Raytheon • Ripe NCC • Sendmail, Inc. • Splunk • Sun Microsystems, Inc.
Taos • Tellme Networks • UUNET Technologies, Inc.

SAGE Supporting Members

Asian Development Bank • FOTO SEARCH Stock Footage and Stock Photography
Microsoft Research • MSB Associates • Raytheon • Splunk • Taos • Tellme Networks

ISBN 1-931971-39-0